

高级大数据人才培养丛书

R语言与大数据编程实战

李倩星 编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书是一本 R 语言入门读物，它旨在帮助读者迅速构建起与数据分析相关的知识体系，并学习如何使用 R 软件实现数据分析方法。无论有无深厚的编程基础或数学基础，本书都能帮助读者成长为一名合格的数据分析师。

本书全面介绍了来自统计分析、机器学习、人工智能等领域的多种数据分析算法，在讲解与之相关的 R 代码时，还讨论了这些算法的原理、优缺点与适用背景。本书按照由易到难的原则组织章节主题，读者将获得最好的阅读体验。通过阅读本书，读者将对 R 语言在数据分析领域的应用有一个全面的认识。这种认识不被特定行业所局限，任何行业的读者都能利用本书介绍的数据分析方法解决本行业的数据分析问题。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

R 语言与大数据编程实战 / 李倩星编著. —北京：电子工业出版社，2017.9

（高级大数据人才培养丛书）

ISBN 978-7-121-32634-9

I . ① R… II . ①李… III . ①程序语言—程序设计 IV . ① TP312

中国版本图书馆 CIP 数据核字（2017）第 215708 号

策划编辑：李 冰

责任编辑：李 冰

特约编辑：彭 瑛 赵海军等

印 刷：三河市华成印务有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮编：100036

开 本：787×1092 1/16 印张：20

字数：512千字

版 次：2017年9月第1版

印 次：2017年9月第1次印刷

定 价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：libing@phei.com.cn。

前言

R 语言是如今最热门的编程语言之一，它由统计学家开发，在解决数据分析问题时具有先天优势。R 语言是一门新兴的语言，掌握它，就是掌握了一门高效的数据分析软件。随着大数据概念的普及，R 语言能够实现的功能越来越丰富，越来越多的数据分析从业人员产生了学习 R 语言的需求。本书迎合时代潮流，讲解了大数据时代下 R 语言渗透最广泛的几个领域，全面介绍了如何使用 R 语言完成数据挖掘工作。对 R 语言编程人员来说，本书是一本不可或缺的工具书。

本书特色

1. 通俗易懂、实用性强，适合各层次读者学习

本书对读者的数学基础或编程基础不做任何要求。在讲解知识点时，本书采用了平实的语言，对每个疑难点都加以详细解释。此外，本书以实用为主旨，秉承“看得懂、学得会、用得上”的编写原则，精心选取了流行于行业前沿的 18 个主题，不仅通俗易懂，还确保读者所学的知识具有实际应用价值。通过阅读本书，读者都能迅速掌握 R 语言的编程技巧及相关的数据分析知识，并在实际工作中立刻应用它们。

2. 条理清晰、结构巧妙，全面盘点数据分析常用算法

数据分析是一个涉及多领域的交叉学科，R 软件的触角同样也能伸展到多个领域。本书选取了统计分析、机器学习、人工智能等多个学科的流行算法作为主题，讲解了如何使用 R 语言实现它们。这些算法有些偏重数学思维，有些偏重编程技巧，本书主要遵循由易到难的顺序排列主题，并尽量把起源于同一学科的算法放在一起。读者可以按照顺序阅读本书，也可以优先选择感兴趣的部分。此外，本书还穿插介绍了与 R 软件相关的一些其他编程主题，这些主题共同形成知识网络，帮助读者迅速成长为能够独当一面的数据科学家。

3. 知识点丰富、可拓展性强，满足读者的多重需求

本书涉及多个学科，全面介绍了 R 软件能够实现的多种算法，满足了读者的三大需求：首先，使用通俗易懂的语言介绍 R 软件，帮助读者实现零基础入门；其次，囊括多种数据分析算法，带领读者全面认识 R 软件的强大之处，帮助读者成长为合格的数据科学家；最后，本书具备较强的可拓展性，从事任何行业的读者都能够从本书中获取适合其行业的知识。本书还给出了 R 语言进阶的线索，无论想向哪一方面进阶，本书都能为读者打造最坚实的基础。

本书内容及体系结构

本书总共 18 章，分别为 R 的基本介绍、原始数据的探索与预处理、R 的数据可视化、R 中参数的估计和检验、R 中的方差分析、R 中的相关分析和回归分析、更高级的数据可视化、R 中的聚类分析和判别分析、R 中的主成分分析和因子分析、R 中的广义线性回归模型、R 中的时间序列模型、R 中的最优化问题、使用 R 绘制地理信息图形、使用 R 构建支持向量机、实现更高效的流程控制和高级循环、R 代码的调试与优化、构建电影评分预测模型、贝叶斯垃圾邮件过滤器模型。这 18 章进一步又分为 5 个部分。

第一部分为本书的第 1~6 章。其中前 3 章展示了 R 软件的一些入门功能，如数据预处理和数据可视化等，后 3 章则介绍了三种基础的统计分析方法，即参数的估计和检验、方差分析、相关分析、回归分析。这 6 章围绕初级的统计方法展开，是数据分析师必备的基本知识。

第二部分为本书的第 7~11 章，这 5 章介绍了更高级的统计方法。其中，第 7 章为第 3 章的延伸，介绍了数据可视化的高级方法，第 8~11 章则介绍了 6 种高级统计分析方法，这部分的内容与第一部分互为补充。

第三部分为本书的第 12~14 章，这部分内容围绕机器学习展开。第 12 章的主题为最优化，是机器学习的基本理论。第 13 章介绍了如何使用矢量化思想绘制地图。第 14 章则介绍了支持向量机，它是最典型的机器学习算法之一。这部分讲解了更高深的 R 语言编程技巧，讨论了一些 R 软件能够解决的高难度问题。

第 15、16 章可视为本书的第四部分。这两章围绕如何优化 R 代码展开，系统地讨论了如何写出错误较少的、运行速度较快的代码。这部分内容帮助读者建立良好的编程习惯，以及与其他 R 用户更好地协同工作。

第 17、18 章则为本书的最后一部分，这两章分别讨论了一个完整的数据挖掘项目。其中电影评分预测的案例着重于表现数据挖掘的完整流程，包括繁复的数据预处理与反复的模型比较等工作；垃圾邮件过滤的案例则引出 R 软件能够处理的另一个主题——文本分析。

上述划分方法仅为一个参考，本书的 18 章既互相联系又彼此独立，读者可按照上述划分方法阅读本书，也可优先阅读某些章节，如将第 3、7、13 章等与数据可视化相关的三个章节放在一起阅读。

本书读者对象

- 想要了解 R 语言的数据分析从业人员。
- 统计学、金融学、计算机技术与科学等专业的学生。
- 想要提高 R 语言编程能力的数据分析师。
- 希望系统学习统计分析方法的从业人员。
- 其他对 R 语言有兴趣的各类人员。

目 录

第 1 章 R 的基本介绍	1
1.1 强大的 R	1
1.2 R 语言在大数据中的应用	2
1.2.1 R 语言用户行为分析	2
1.2.2 R 语言处理金融大数据	3
1.2.3 R 语言天气数据可视化	4
1.2.4 R 语言医疗大数据分析	4
1.3 R 的安装与启动	5
1.3.1 安装并启动 R	6
1.3.2 安装并启动一个 IDE	7
1.4 R 的向量、矩阵和数组	8
1.4.1 向量的操作方法和固有属性	8
1.4.2 矩阵的操作和运算	10
1.4.3 数组中的维度函数	13
1.5 R 的列表和数据框	14
1.5.1 列表的特性和编辑方法	14
1.5.2 数据框的创建和基本操作	17
1.6 R 数据文件的载入和载出	19
1.6.1 结构化纯文本文件的读取和输出	19
1.6.2 其他文件的读取和输出	22
1.7 向 R 中安装包	23
第 2 章 原始数据的探索与预处理	26
2.1 度量数据集的集中程度	26
2.2 度量数据集的分散程度	27
2.2.1 极值、方差和标准差	27
2.2.2 标准误和偏度系数、峰度系数	29

2.3	创建一个数值摘要表.....	30
2.4	异常值的观测与说明.....	32
2.4.1	利用箱线图观测异常值并处理.....	32
2.4.2	异常值检测的其他情况和说明.....	34
2.5	缺失值的填补与处理.....	35
2.5.1	删除缺失值或对其进行简单填补.....	36
2.5.2	按照相关性对空缺值进行填补.....	38
第 3 章	R 的数据可视化	40
3.1	plot() 函数和常用的图形参数.....	40
3.1.1	设置 plot() 函数中的参数.....	40
3.1.2	修改散点图的坐标并加入标注.....	43
3.2	经典的基础图形及用途.....	45
3.2.1	线图.....	45
3.2.2	直方图.....	49
3.2.3	箱线图和茎叶图.....	52
3.3	将图形组合起来.....	55
3.4	更多的高水平作图函数.....	57
3.5	更多的常用作图命令.....	59
第 4 章	R 中参数的估计和检验.....	62
4.1	使用 R 进行点估计和区间估计.....	62
4.1.1	简单的点估计和区间估计.....	62
4.1.2	估计单侧置信区间.....	65
4.2	与正态总体有关的参数检验.....	68
4.3	列联表与独立性检验.....	71
4.4	几种检验数据分布的函数.....	72
4.5	对非正态总体的区间估计和检验.....	75
4.5.1	非正态总体的区间估计.....	75
4.5.2	非参数检验中的符号检验.....	76
4.5.3	非参数检验中的秩检验.....	78

第 5 章 R 中的方差分析	80
5.1 方差分析模型的建立	80
5.2 单因素方差分析	81
5.2.1 单因素方差分析的数学思想与模型	81
5.2.2 检验样本是否满足方差分析的假设条件	82
5.2.3 构建单因素方差分析模型	84
5.3 多因素方差分析	87
5.3.1 多因素方差分析的数学思想与模型	87
5.3.2 不考虑交互作用的双因素方差分析	88
5.3.3 考虑交互作用的双因素方差分析	89
5.4 秩检验和协方差分析	91
5.4.1 对控制变量应用秩检验方法	91
5.4.2 协方差分析的假设与应用	92
第 6 章 R 中的相关分析和回归分析	94
6.1 多种相关系数的度量和分析	94
6.1.1 简单相关系数的计算和检验	94
6.1.2 散布矩阵图和偏相关系数	96
6.1.3 典型相关分析	98
6.2 线性回归分析及其常规参数	99
6.2.1 对数据进行预处理	100
6.2.2 构建第一个回归模型	101
6.2.3 修正方程并检验残差	102
6.3 使用逐步回归筛选自变量	104
6.3.1 逐步回归的思想与分类	104
6.3.2 构建逐步回归模型	105
6.4 哑变量和逻辑回归	107
6.4.1 哑变量和逻辑回归的思想	107
6.4.2 向线性回归模型中纳入哑变量	108

第 7 章 更高级的数据可视化	110
7.1 基础图形的拓展与延伸	110
7.1.1 绘制分类散点图并添加图标	110
7.1.2 绘制含多种类别的密度分布图	112
7.1.3 复合条形图和堆栈条形图	114
7.2 有关多元分布函数的特殊图形	117
7.2.1 星图和脸谱图	117
7.2.2 轮廓图	120
7.2.3 调和曲线图	122
7.3 建立最简单的 3D 图形	123
7.4 如何让图形更美观	125
7.5 更多的绘图包和系统	128
第 8 章 R 中的聚类分析和判别分析	129
8.1 几种聚类分析的异同	129
8.2 使用 R 实现 KNN 聚类	130
8.2.1 KNN 算法的思想和模型	130
8.2.2 使用 R 实现 KNN 聚类	131
8.3 使用 R 实现系统聚类	133
8.3.1 系统聚类的思想和模型	133
8.3.2 使用 R 实现系统聚类	134
8.4 使用 R 实现快速聚类	136
8.4.1 快速聚类的思想和模型	136
8.4.2 使用 R 实现快速聚类	137
8.5 几种判别分析模型综述	140
8.5.1 距离判别模型	140
8.5.2 Fisher 判别模型	142
第 9 章 R 中的主成分分析和因子分析	145
9.1 主成分分析的实现与应用	145
9.1.1 主成分分析的模型假设和数据处理	145
9.1.2 构造一个主成分分析模型	147

9.1.3 计算主成分的综合得分	149
9.2 因子分析的初次构建与完善	150
9.2.1 构造一个简单的因子分析模型	150
9.2.2 计算因子得分并分析	152
9.3 对因子分析模型进行修正	153
9.3.1 修改因子分析模型中的因子个数	153
9.3.2 基于主成分法和主轴因子法进行因子分析	155
9.4 在降维分析的基础上进行回归分析和聚类分析	157
9.4.1 在降维分析的基础上进行回归分析	157
9.4.2 在降维分析的基础上进行聚类分析	160
第 10 章 R 中的广义线性回归模型	162
10.1 一般的广义线性回归模型	162
10.1.1 使用二次函数拟合线性回归模型	162
10.1.2 拟合更多的广义线性模型	164
10.1.3 比较线性模型的优劣	166
10.2 Logistic 线性回归模型	168
10.2.1 Logistic 模型的原理与构建方法	168
10.2.2 Logistic 模型的显著性检验和优势比	170
10.2.3 修正被警告的 Logistic 模型	171
10.3 泊松回归分析模型	173
10.3.1 拟合第一个泊松回归模型	174
10.3.2 泊松回归模型的过散布检验	176
10.4 广义线性模型的交叉验证	178
第 11 章 R 中的时间序列模型	180
11.1 将数据转换为时间序列格式	180
11.1.1 使用 ts() 函数转换数据格式并绘制时间序列曲线	180
11.1.2 使用 zoo() 函数转换数据格式并绘制时间序列曲线	182
11.2 分解时间序列并检验时间序列的自相关性	185
11.2.1 使用经典方法分解时间序列	185
11.2.2 使用 STL 方法分解时间序列	186

11.3	探究时间序列的自相关性	188
11.3.1	使用月图和季度图探究自相关性	188
11.3.2	使用散点图探究自相关性	189
11.4	构建时间序列并预测	191
11.4.1	均值预测、单纯预测和漂移	192
11.4.2	不考虑长期趋势和季节波动的简单指数平滑	195
11.4.3	在指数平滑中加入长期趋势和季节波动	196
11.4.4	自回归移动平均模型	198
第 12 章	R 中的最优化问题	201
12.1	最优化问题简述	201
12.2	黄金分割法	202
12.2.1	黄金分割法和局部最优解	202
12.2.2	使用 R 实现黄金分割法	203
12.3	牛顿最优化方法	205
12.3.1	牛顿法的算法原理	206
12.3.2	在一维情形下实现牛顿迭代法	207
12.3.3	在多维情形下实现牛顿迭代法	209
12.4	最快上升法	210
12.4.1	利用梯度求解上升最快的相邻点	210
12.4.2	构建最快上升法函数并检验	212
12.5	R 中的最优化函数	213
第 13 章	使用 R 绘制地理信息图形	216
13.1	绘制世界、国家、省市地图	216
13.1.1	使用 map() 函数绘制地图	216
13.1.2	另一种绘制地图的方法	217
13.1.3	分省市绘制地图	218
13.2	向地图中添加颜色	220
13.2.1	向地图中添加颜色前的准备工作	220
13.2.2	在地图上添加颜色	221
13.3	向地图中添加标签和线条	222
13.3.1	向地图中添加标签前的准备工作	222

13.3.2	在地图上添加标签	224
13.3.3	在地图上添加线条	225
13.4	使用其他格式的文件优化地图	226
第 14 章	使用 R 构建支持向量机	230
14.1	构建一个简单的支持向量机	230
14.1.1	支持向量机的算法原理	230
14.1.2	构建一个简单的支持向量机	232
14.1.3	使用其他核函数构建支持向量机	235
14.2	优化支持向量机的参数	237
14.2.1	优化参数 degree	238
14.2.2	优化参数 cost	241
14.2.3	优化参数 gamma	243
14.3	比较支持向量机与 Logistic 回归的优劣	246
14.4	比较支持向量机和 KNN 聚类算法的优劣	249
第 15 章	实现更高效的流程控制和高级循环	251
15.1	R 中的流程控制	251
15.1.1	if 语句的多种实现方法	251
15.1.2	ifelse 语句与花括号的结合	252
15.1.3	适合多分支情况的 switch 语句	254
15.2	R 中的 for 循环、while 循环和 repeat 循环	256
15.2.1	R 中的 for 循环和 while 循环	256
15.2.2	R 中的 repeat 循环	258
15.3	apply 家族中的循环函数	260
15.3.1	R 中的 apply() 函数	260
15.3.2	R 中的 lapply() 函数和 sapply() 函数	263
15.3.3	R 中的 tapply() 函数	265
15.3.4	R 中的 mapply() 函数	268
15.4	更多的高级循环函数	270
15.4.1	R 中的 replicate() 函数和 sweep() 函数	270
15.4.2	R 中的 aggregate() 函数	273

第 16 章 R 代码的调试与优化.....	276
16.1 R 代码的常见信息与警告	276
16.1.1 R 代码的正常信息与警告	276
16.1.2 R 代码中的警告处理方法	278
16.2 R 代码中的错误与错误处理方法	279
16.2.1 使用 try() 函数处理错误信息	279
16.2.2 将 try() 函数与循环相结合	281
16.3 调试 R 代码	282
16.3.1 查看调用栈或暂停代码	282
16.3.2 修改 error 选项	284
16.4 向量化编程方法	285
16.4.1 向量化编程思想	285
16.4.2 比较循环和向量的运行速度	286
第 17 章 构建电影评分预测模型	289
17.1 获取数据并探索	289
17.2 利用 recommenderlab 包处理数据	291
17.3 建立模型并评估	293
17.3.1 模型的选择与建立	293
17.3.2 模型之间的比较和评估	295
第 18 章 贝叶斯垃圾邮件过滤器模型	297
18.1 贝叶斯模型中的条件概率	297
18.2 复杂的数据预处理过程	298
18.2.1 利用 for 循环读入多封邮件正文	298
18.2.2 利用 tm 包进一步转换数据格式	300
18.2.3 将 TDM 转换成真正有用的数据框	301
18.3 利用 occurrece 值构造分类器	303
18.3.1 完成理论准备并处理测试邮件和普通邮件	303
18.3.2 创建一个函数用于比较概率	305

第1章 R 的基本介绍

作为一门新兴的编程语言，R 是如今值得学习的语言。由统计学家开发出的 R 语言具有许多奇特性质，本章将较为全面地介绍 R 的特性和用途，并讲解 R 的安装方法、变量类型、从其他数据源读取数据、程序包等基本知识。本章帮助读者对 R 形成整体印象，同时本章内容也是后续章节的基石。

1.1 强大的 R

R 语言脱胎于 S 语言，是一门专门用于处理数据探索、统计分析等任务的编程语言。它由统计学家开发完成，在数据分析方面具有天然的优势，运行 R 程序的 R 软件是如今最流行的统计软件之一。

与其他统计软件相比，R 软件最特别的地方在于它是开源的。这同时意味着：第一，R 是免费的；第二，R 的用户能够自由地参与到 R 的开发中。R 社区将它的忠实用户聚合在一起，这些用户主要由统计学家、计算机学家、数据分析师等组成，不同领域的用户在 R 社区中交流碰撞，协助 R 核心团队丰富和完善 R 的功能。

R 的用户之间具有非常紧密的联系，他们最大的贡献是创建了形形色色的程序包，这些程序包分别封装了一些具有特定作用的函数。如今，R 软件已经内置了非常丰富的各类函数库，能够满足绝大多数统计人员的各类需求，它的制图功能也远超其他统计软件。

R 的另一个特点在于它支持混合型的编程范式。R 是一种解释型的语言，当用户在 R 软件中编写好一条代码后，R 会立即执行它。这种做法的好处在于用户可以即时地看到程序的返回结果，在作图时尤其方便。R 是一种面向对象的语言，同时它也支持函数式编程，即用户可以在 R 中调用现成的或自己编写的函数，这一点与 C 语言较为相似，但 R 要比 C 语言更加灵活。

尽管 R 的优点很突出，但它也同样具有局限之处。首先，R 语言的编程原理较为传统，在处理数据时，R 需要将数据全部载入内存，这一点极大地影响了 R 的运行效率，尽管如今的计算机内存做得越来越大，但在有些大规模数据集的处理工作中，R 还是会显得不够得力。其次，R 软件的保密性不如 SAS 等统计软件好，这限制了 R 在大型商业项目中的应用。最后，由于 R 软件是由统计学家开发的，因此其语法设计并不特别严谨，有时它会出现一些奇怪的错误。

随着大数据时代的到来，R 语言正被越来越多的人关注，不仅是统计分析和数据挖掘，一些研究机器学习和模式识别的专家同样关注到 R 的发展。根据 TIOBE 提供的编程语言排行榜，R 语言的流行程度在近几年内已经飙升至前十名，其火爆程度只有 Python 才能与其比肩，而同为统计软件的 SAS 和 MATLAB 则一直徘徊于二三十名的位置。

R 的优点使它广泛地流行于统计人员和中小型商业公司中。Google、百度等互联网巨头则将 R 语言看作一个沙盘，使用软件验证各种数据模型的可行性，并最终使用其他语言实现。随着 R 的用户越来越多样化，其可扩展能力进一步强化，能够解决的问题也越来越丰富。如今，金融、医药、教育、社会科学等每一个需要数据分析的领域都需要精通 R 的人才。

1.2 R 语言在大数据中的应用

R 语言的起源是统计学家为解决数据分析领域问题而开发的语言，所以和 MATLAB、Python 等可用于数据处理的语言相比，在数据分析处理方面具有一些独特优势，本节将讲述 R 语言在大数据领域中的典型应用。

1.2.1 R 语言用户行为分析

近几年，淘宝、京东等几家电商的价格战打得不亦乐乎，而从电商发布的战果来看，几败俱伤的价格战已经无法保证电商的利润，他们开始转向利用大数据分析工具对用户行为进行分析，通过对大数据的充分使用和挖掘在商战中获胜。

何为“用户行为分析”？简单的理解就是对用户在网站上发生的所有行为进行分析，找到里面的规律和用户感兴趣的信息。这些行为如搜索商品信息、浏览新闻、购物评价和打分、美团点评、加入收藏列表、加入购物车、购买、使用特价购物券、换货和退货等；除此之外，还包括在第三方网站上的相关行为分析，如比价、看相关评测、参与讨论、社交媒体上的交流、与好友互动等。电子商务的突出特点就是可以通过后台收集到大量客户在购买前的行为信息，而这些信息对于分析用户行为至关重要。

全球电子商务的创始者之一亚马逊公司结合大数据分析工具，以迅雷不及掩耳之势，彻底颠覆了很多行业的市场规则及竞争关系。亚马逊取胜的根本原因在于对数据的战略性认识和使用，亚马逊通过传统门店无法比拟的互联网手段，获取了极其丰富的用户行为信息，并且进行深度分析与挖掘。

电商通过对用户行为的分析，可以制定更加贴心的服务。例如，当客户浏览了多款手机而没有做购买的行为时，系统会把适合客户的品牌、价位和类型的多款手机信息推送到客户的账户，只要客户登录系统，就可以看到这些推送信息。这样的个性化推荐服务往往会起到非常好的效果，不仅可以提高客户购买的意愿，缩短购买的路径和时间，还可以在比较恰当的时机捕获客户的最佳购买冲动，提升用户体验，是一个一举多得的好方式。

在电商领域中，用户的行为信息量之大令人难以想象，据不完全统计，用户在电商网站购买一个商品前，平均会浏览 3 ~ 5 个网站（淘宝、京东、1 号店等）、30 ~ 36 个页面，统计起来对于一个一天有近百万访问量的中型电商，一天就会有 1TB 左右的活跃数据。

除了电商，爱奇艺、优酷、土豆网等各类视频网站已成为人们娱乐和学习的重要平台，视频网站成功与否的最重要衡量标准就是用户的满意度。由于 Web 应用能够以很细的粒

度、很高的频度不断记录用户的行为轨迹，这些数据中隐藏着用户的访问习惯、兴趣偏好及情绪变化等信息，同时也隐含着用户群体行为的规律和发展趋势。挖掘深藏在数据背后的知识，能够发现用户习惯的观看流程，访问网站的时间及喜爱的视频，各视频间存在的关联关系等。掌握了这些知识，就能科学解决用户跳出等问题，根据用户访问习惯改进网站服务流程，以及针对用户进行个性化服务，变革传统的网站管理和运营模式，主动提升用户的体验以促进视频网站的快速发展。

综观国内外成功的电商和视频网站等互联网企业，对用户行为信息的分析和使用，无不在这个必争之地做了大量投入。他们对数据战略性的高度认识和使用，非常值得国内的电商学习和借鉴。

R 语言作为能够进行交互式数据分析和探索的强大平台，具备一套完整的数据处理、计算和制图软件系统，在电商用户行为数据分析和挖掘领域应用广泛。基于 R 语言对视频网站的用户在线行为数据进行分析挖掘，通过对采集到的用户行为数据进行引流指标、黏性指标的分析和挖掘，可以得到网页被访问的频繁度、停留时间、用户观看视频的喜好等信息。将分析的结果应用到网站运营与管理中，不仅能够为网站个性化服务、精准推销和开发新型业

务模式提供技术和理论支撑，而且能够把握当前网络关注的热点问题，从而正确引导用户的网络舆论方向。

1.2.2 R 语言处理金融大数据

最近几年，数据分析方法在商业和金融市场上的重要性持续增加，因为我们有丰富的数据环境，经济和金融市场的历史数据相比以前更加综合和完整，在许多国家成百上千个变量的数据可以更系统、更精确地搜集，计算处理上的便利和统计软件包的使用使得对复杂的高维金融数据的分析成为可能，通过互联网可以很容易地应用开源软件包下载公开的金融数据，如 R 语言和软件开发环境，所有这些软件的特性和功能免费公开，因而被广泛使用。

2008 年的金融危机在某种程度上是由错误的金融模型造成的，既有模型过于简单的原因，又有模型过于复杂的原因，房地产经纪人和买家依赖于一个隐式模型，它表明价格已经在高位，且还会继续上涨。贷款人使用统计模型来对打包的按揭产品进行分析设计，这似乎可以奇迹般地降低风险，然而最后的结果是灾难性的，在 4 年之后仍然可以感受到房地产泡沫的影响。

那么，如何进行有用的并且没有危险的金融分析呢？首先应该对金融数据有一个基本的理解，尤其是时间序列数据，因为不确定性是主要的影响因素，比如可以用概率模型来描述资产收益率的频率分布，利用时间序列模型对数据进行描述平滑和季节调整。

使用 R 语言进行金融分析时，可以利用 R 语言的优势，研究分析基于 Hadoop 存储证券的日内交易数据，通过 RHive 连接 R 语言与 Hive，建立相关性算法模型，在历史数据中回测，构建投资决策组合，并生成可视化结果用于展示。

R 语言 `quantmod` 包是股市金融建模应用比较多的一个包。该包从多个数据源获取历史数据，绘制金融数据图表，以及在金融数据图表中添加各种技术指标，通过多种金融

模型分析，辅助股票筛选和判断。该包获取数据的来源主要有两个：Yahoo! 和 Google。最常用的是 Yahoo! 中的各种数据。但该包只能获取股票的历史交易记录信息，如最大值、最小值、开盘价、收盘价及成交量。在此基础上利用股票的历史数据，通过 R 语言建立模型，并对数据进行分析，从模型的检验决定未来的交易行为。

1.2.3 R 语言天气数据可视化

R 语言天气数据可视化，就是通过获取天气的历史大数据，使用 R 语言和可视化技术，展示中国每个省份的天气情况，给准备旅游的朋友提供一种出行提示。除了旅游，在传统零售行业，雨天大概会影响相对于晴天 30% ~ 40% 的销售业绩，所以从网上获取天气数据进行分析，并根据天气数据做出预测，提前做好预防措施和提醒业务人员，把损失减少到最低就显得十分重要。要实现天气数据可视化，需要实现的功能和遇到的问题如下。

- 天气数据：数据从哪里找到；如何下载；如何存储。
- 定时任务：天气数据需要每日更新，图片需要每日新生成。
- 地图和天气可视化：要把中国行政区划图和天气数据（包括风力方向可视化）结合在一起绘图，让用户一眼就能看明白。
- Web 展示：通过可视化技术，我们生成的只是一张静态图片，要如何发布到 Web 端进行展示。
- 微博：结合新浪微博，让更多的用户看到并使用这个应用。
- 用户交互：用户可以查看不同日期、不同类型的图片，用户还可以通过微博分享。

从上面的描述中，单独使用一种语言不容易实现这些功能。单独用 PHP 开发，做一个 Web 网站非常容易，连接新浪微博也有现成的 SDK 可以调用，获取数据及存储也不麻烦，但是如何实现地图和天气数据的可视化？这个是 R 语言的强项，所以，可以将 R 语言和 PHP 语言相结合，发挥 R 语言的优势，用 R 语言的 rvest 包就可以方便获取天气数据，并实现天气数据的可视化。

1.2.4 R 语言医疗大数据分析

医疗大数据是相对于一般数据而言的，指的是人们从医疗系统的软件系统中捕捉大容量数据，通过大数据分析获得新的认知，从而创造新的价值来源。医疗大数据几乎包含公民所有个人信息，包括医疗、饮食、住所、旅行登记等。在临床操作方面，有 5 个主要场景的大数据应用。根据麦肯锡公司的估计，如果这些应用被充分采用，针对美国一个国家的医疗健康开支一年就将减少 165 亿美元。主要场景的大数据应用包括 5 个方面。

(1) 比较效果研究。

通过全面分析病人特征数据和疗效数据，然后比较多种干预措施的有效性，可以找到针对特定病人的最佳治疗途径。世界各地的很多医疗机构，如德国 IQWiG（德国医疗质量和效率研究所）、加拿大普通药品检查机构等，已经开始了 CER（比较效果研究）项目，并取得了初步成功。

（2）临床决策支持系统。

临床决策支持系统可以提高工作效率和诊疗质量。大数据分析技术将使临床决策支持系统更智能，这得益于对非结构化数据的分析能力日益加强。比如，可以使用图像分析和识别技术，识别医疗影像数据，或者挖掘医疗文献数据建立医疗专家数据库，从而给医生提出诊疗建议。此外，临床决策支持系统还可以使医疗流程中大部分的工作流向护理人员 and 助理医生，使医生从耗时过长的简单咨询工作中解脱出来，提高工作效率。

（3）医疗数据透明度。

提高医疗过程数据的透明度，可以使医生、医院的绩效更透明，间接促进医疗服务质量的提高。数据分析可以带来业务流程的精简，降低成本，找到符合需求工作更高效的员工，从而提高护理质量，给病人带来更好的体验，也给医疗服务机构带来业绩增长潜力。公开发布医疗质量和绩效数据还可以帮助病人做出更明智的健康护理决定，这也将使医院提高总体绩效，更具竞争力。

（4）远程病人监控。

对慢性病人的远程监控系统收集数据，并将分析结果反馈给监控设备（查看病人是否遵从医嘱进行治疗），从而确定后续用药和治疗方案。通过对远程监控系统产生的数据分析，可以减少病人的住院时间，减少急诊量，提高家庭护理比例和门诊医生预约量。

（5）对病人档案的先进分析。

在病人档案方面应用大数据分析可以确定某类疾病的易感人群。帮助识别哪些病人有患糖尿病、高血压等疾病的高风险，使他们尽早接受预防性保健方案。这些方法也可以帮助患者从已经存在的疾病管理方案中找到最好的治疗方案。

R 语言医疗大数据分析的主要目标是针对海量医疗数据分析，提出了医疗大数据分析的新模式，并分析比较了两种分析模式的区别，开展 R 语言等统计模型的应用研究，建立从统计模型、指标提取、统计模型检验与优化、统计模型重写、数据可视化的一套方法，在此基础上具体针对 BI 工具发现问题，应用 R 语言进行了异常医疗质量指标的相关因素分析。

利用 R 语言对门诊和用药大数据进行分析，利用信息技术探索基层医疗卫生机构门诊用药规律，为合理用药提供依据。具体做法是通过省基层医疗机构管理信息系统，提取医院门诊某阶段的用药情况。通过 `arules` 包中的 `eclat` 函数，设置参数最小求频繁项集，即在门诊用药记录中，某种药品的出现频率，按照由大到小的次数排序，得到门诊使用较多（频繁）的药品列表；通过 `arules` 包中的 `apriori` 函数，可以发现在门诊用药数据中的多条关联规则。

1.3 R 的安装与启动

本节介绍 R 软件的安装与启动，以及几个好用的 IDE，再比较使用 IDE 执行代码与直接使用 R 控制台执行代码的异同。

1.3.1 安装并启动 R

R 核心团队免费向用户提供 R 软件，无论用户使用的是 UNIX、Windows 还是 Mac OS 系统，都可以在 R 网站上很方便地下载最新的 R 软件。

R 网站的主页网址为 <https://www.r-project.org/>，这是可直接登入的英文网页，在这一主页中使用蓝色字体标出了一些超链接，分别链向与 R 相关的其他网页，比如 FAQ（常见的问题及答案）网页，以及有关 R 的新闻等。

在 R 主页的 **Getting Started** 标题下方有一个蓝色加粗的 **download R**，它指向一个 CRAN 的镜像网址汇总页，汇总了分布在全球各个国家的许多镜像服务器，其中我国至少有 4 个，用户可以选择最近的镜像。在镜像页中，用户根据自己的计算机系统选择对应的 R 软件版本，即可点击下载。R-3.4.1 是目前最新的版本，Windows 版本大约有 62.2MB 大小，此时下载的 R 仅包含最基础的函数，在后续的学习中还需要陆续添加其他的程序包。

执行下载好的 exe 文件，Windows 系统中将弹出一个安装向导，R 软件的安装目录默认为系统盘，但将其安装到其他盘也不影响使用。需要注意的是在第四步中，安装向导要求用户选择组件时，最好不要选择默认的选项。

如图 1.1 所示，当安装向导执行到第 4 步时，用户组件共有 4 个组件可供选择，默认选项是将这 4 个组件全部安装。在“用户安装”下拉框下还有“32 位用户安装”、“64 位用户安装”、“自定义安装”三个选项，用户只需根据自己的机型选择 32 位或 64 位即可，不需要将 4 个组件全部下载。

R 成功安装完毕后，桌面将出现一个宝蓝色的 R 图标，只需双击 R 图标，即可启动 R 软件。

Windows 系统下运行 R 的工具也称为 RGui，即 R 图形用户界面。图 1.2 所示是一个 64 位的 RGui。在 RGui 的菜单栏中有一些中文选项及一些快捷操作，其中最常用的菜单选项是文件菜单和程序包菜单。

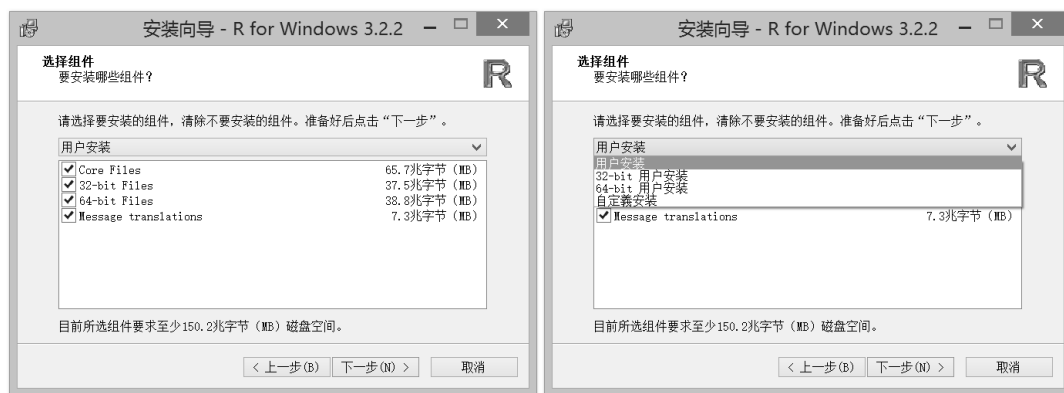


图 1.1 选择 R 软件的组件

在 RGui 中间还有一个更小的 R Console，也称为 R 控制台，控制台是用户执行代码的地方。蓝色的文字对 R 进行了一些基本声明，并给出了一些用以查看帮助文件的命令。

在声明文字下方有一个醒目的红色“>”形提示符,其后跟随了一个闪烁的红色光标。“>”提示用户在此输入命令,只需回车,命令就会被 R 执行,同时 R 会在新的一行上再次生成“>”提示符。显然,退出 RGui 时,只需点击右上角红色小叉即可。

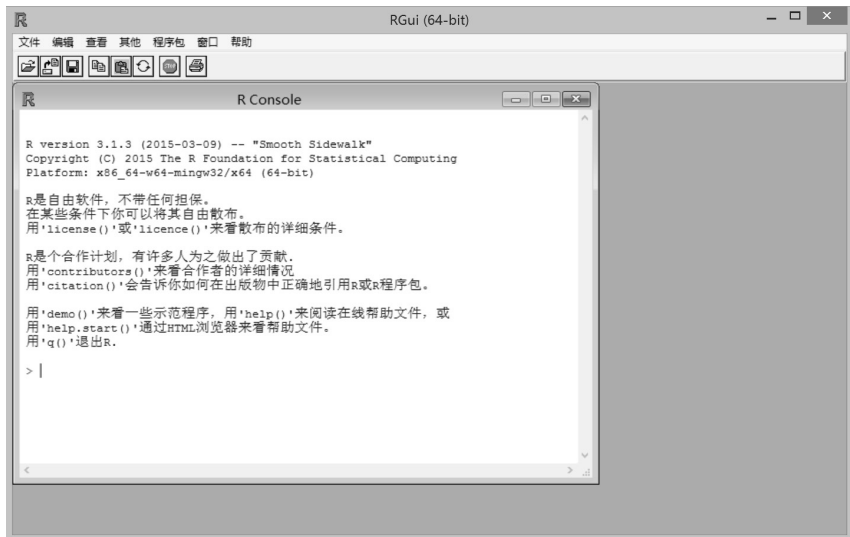


图 1.2 R 控制台

在 Mac OS 环境下运行 R 的工具是 R.app, 在 UNIX 环境下启动 R 时需要在路径中包含 R 文件, 然后输入命令 R 即可。

1.3.2 安装并启动一个 IDE

R 软件虽然提供了文本编辑器, 但为了更方便地使用 R, 大多数用户都会选择额外安装一个 IDE (集成开发环境) 用于辅助编程。IDE 提供一个图形开发环境, 语法编辑功能也通常更为强大。

在此向统计人员强烈推荐 RStudio 软件。RStudio 是一个专门为 R 定制的免费 IDE, 它将 R 中的特色功能体现得淋漓尽致。网址 <https://www.rstudio.com/products/rstudio/download/> 提供了适用于不同计算机平台的 RStudio 版本, 包括 Windows、Mac OS、Ubuntu、Fedora 等, 用户只需在“Installers for Supported Platforms”标题下方选择合适的版本即可。

RStudio 软件的安装十分简单, 仅需设定安装目录和文件夹名称即可。为了方便起见, 通常把 R 和 RStudio 放在同一个目录下。安装完毕后, 桌面上同样会生成一个宝蓝色的、与 R 稍有区别的 RStudio 图标, 双击图标即可启动 RStudio 软件。

如图 1.3 所示, RStudio 软件被切分为 4 个小窗口。其中, 左上角的窗口用于展示 R 脚本, 用户可以在这里打开已经写好的 R 脚本, 或者在这里编写一个脚本而不立刻执行它。左下角的窗口则是 R 软件的控制台, 它其实就是 1.2.1 节中提到的 R 控制台, 在用户打开 RStudio 时, RStudio 会在后台启动 R 软件。

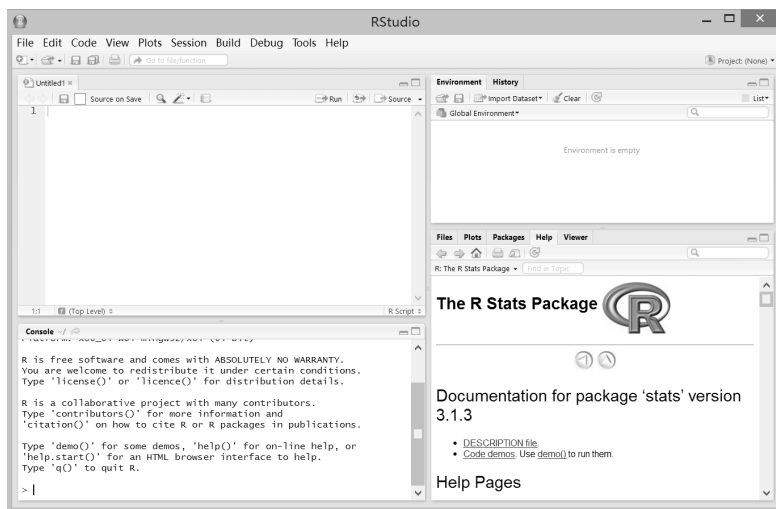


图 1.3 RStudio 软件启动界面

RStudio 的右上角显示了环境栏和历史栏，在执行代码时，环境栏会自动显示当前程序中都有哪些变量，以及它们的属性分别是什么，当变量繁多时这一栏格外有用。历史栏中则存有已经执行过的历史代码。右下角的窗口同样提供了好几个下拉菜单，其中包括 R 中的各类帮助、函数介绍和程序包介绍。

总的来说，RStudio 能够提供的信息远比 R 软件丰富，它提供了更加美观的绘图窗口，用户在查看帮助时也显得尤为便捷。此外，RStudio 也支持远程访问，用户可以在平板电脑或手机上打开它，并远程访问计算机上的 R 执行代码。

RStudio 的唯一缺点在于，它只用于 R 开发。因此，对并非只写 R 代码的程序员来说，RStudio 也许不是最好的选择。不过有许多支持 Java、C、Python 等语言的 IDE 同样也支持 R，其中较为流行的优质 IDE 有 Emacs、Eclipse、Tinn-R 等，用户完全可以根据自己的喜好选择一个最合适的 IDE。

与 RStudio 相比，这些支持多种语言的 IDE 在 R 的特色功能方面体现得不如 RStudio 好，但它们能够提供诸如语法高亮、快捷操作等实用的功能。值得注意的是，有时用户需要下载专门的插件或下载专门的版本才能使 IDE 支持 R。

1.4 R 的向量、矩阵和数组

向量、矩阵和数组是 R 中的三种基本变量，也是数据分析中最常见的数据存储单位。本节讨论这三种变量的特点，并展示它们的操作方法。

1.4.1 向量的操作方法和固有属性

向量是一个一维变量，即一系列数据的集合。它是在数据分析时使用的最小的单位，矩阵和数组都是向量的扩展结果。我们通常使用 `c()` 函数来创建向量，其他可以用于创建向量的函数还有 `seq()` 函数和 `rep()` 函数，其中前者用于排序，后者用于重复。


```

> x <- c(1:5)
> c(6:10) -> y
> x;y
[1] 1 2 3 4 5
[1] 6 7 8 9 10
> x[0];y[1]
Integer(0)
[1] 6

```

上述代码创建了两个向量并分别将它们赋给了 x 和 y 。第一行代码中 `c()` 函数内的冒号表示创建一个序列，“<-”符号则将这个序列赋给了 x 。第二行代码使用“->”符号将另外一个序列赋给了 y 。这两条赋值命令都是合法的，不过“<-”符号是较常见的选择。

R 并不会主动返回 x 和 y 的结果，为了查看 x 和 y 中存放的内容，我们执行了第三条代码。第三条代码使用分号隔开了 x 和 y ，因此 R 会同时返回这二者。显然， x 中存放了序列 1,2,3,4,5， y 中存放了序列 6,7,8,9,10。此外，也可以使用下标查看向量中的某个具体元素，R 中向量的下标从 1 开始标起，这一点与 C 语言等其他语言是不同的。第 4 条代码查看了向量 x 中下标为 0 的元素和向量 y 中下标为 1 的元素，可以发现， $x[0]$ 返回了一个奇怪的结果， $y[1]$ 则返回了 y 中第一个元素。

```

> z <- c(x,y,11,12)
> z
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> z <- x+y
> z
[1] 7 9 11 13 15
> names(z) <- c("a","b","c")
> z
  a    b    c <NA> <NA>
7   9  11  13  15

```

`c()` 函数同样可以用于拼接向量，它既可以将已有的向量拼接起来，也可以将已有的向量和新的数字拼接起来。上述代码中第一条代码将向量 x 、 y 和数字 11、12 拼接成一个更长的向量 z ，第二条代码展示了向量 z 中存储的内容。

第三、四条代码将向量 x 和 y 相加后的值赋给了向量 z ，并查看了 z 。由于 x 和 y 的长度相同，因此 z 中存储的内容即为 x 和 y 中元素一一对应相加后的结果。R 中可以直接在向量上执行加、减、乘、除等基本运算，但需要保证执行运算的两条向量长度相等，否则将得到难以解释的或错误的结果。在执行了第 3、4 条代码后，向量 z 中的内容已经被替换为了序列 7,9,11,13,15。

我们不仅可以为向量命名，向量中的每一个元素也可以单独得到名字。第 4 行使用 `names()` 函数为向量 z 中的元素命名，双引号引起的 `a`、`b`、`c` 表示这三个字母在这里是字符形式，当然，它们也不可能是别的形式。由于我们只命名了三个向量，因此向量 z 中只有前三个元素有名字，后两个元素的名字为 `<NA>`，也就是空的意思。为元素命名和为向量命名遵循一样的规则，即名字中只能含有字母、数字、点和下画线，但不能以数

字或一个点后跟数字开头，也不能和 R 中的保留字相同，比如 if 和 for。

```
> min(z)
[1] 7
> range(z)
[1] 7 15
> sum(z)
[1] 55
> var(z)
[1] 10
```

作为一个统计软件，R 中内置的运算函数非常丰富，写起来也非常方便。上述代码给出了 min() 函数、range() 函数、sum() 函数和 var() 函数 4 个函数，它们的功能分别是对 z 取最小值、给出 z 的范围、计算 z 的和、计算 z 的方差。

其他经常用到的向量运算函数还有用于取出数据集中最大值的 max() 函数，求最值元素的下标数的 which.min() 函数和 which.max() 函数，求向量中元素连乘积的 prod() 函数，求均值、标准差、顺序统计量的函数，求积分的函数等。这些函数不但可以用于向量运算，也可以仅对向量中的某些元素进行运算。

```
> length(z)
[1] 5
> mode(z)
[1] "numeric"
> z <- as.character(z)
> mode(z)
[1] "character"
> min(z)
[1] "11"
```

与向量相关的另一个重要知识是向量的属性。我们同时关心向量的长度和类型。length() 函数用于查看向量的长度，由上述代码可知，向量 z 的长度是 5。mode() 函数用于查看向量的类型，显然，向量 z 的类型是 numeric 型，即数值型。第三行代码使用 as.character() 函数将向量 z 的类型转换为字符型，其他类似函数还有 as.factor() 函数、as.integer() 函数等。此时再次查看向量 z 的类型，其返回结果显示 z 已变为了字符型。

向量的类型与其适用的运算函数有关，当向量 z 变为字符型后，再次查看它的最小值，此时出现的结果为 11，显然不是我们想要的结果。通常无须声明新向量的类型，R 就会自动为新向量分配一个合适的类型。而向量的类型转换也不是没有限制，比如数值型的向量可以转为字符型，但存储了字母元素的字符型向量就无法转化为数值型向量。

1.4.2 矩阵的操作和运算

在向量、矩阵、数组中，矩阵是最具价值的一类变量。矩阵可以看作向量的拓展，向量是只具有长度的一维变量，而矩阵则是同时具有“长度”和“宽度”的二维变量，在实际的数据分析工作中，许多数据都以矩阵的形式进行处理。

```
> x <- c(1:12)
> matx1 <- matrix(x,nrow=3)
> matx1
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

不妨从矩阵的创建开始学习矩阵的特性。上述代码中第1行代码创建了一个1~12的数字序列 x ，第2行代码则利用 `matrix()` 函数创建矩阵 `matx1`。在第2行代码中向 `matrix()` 函数传入了两个参数，第1个参数设定使用 x 的值创建矩阵，第2个参数则指定矩阵的行数为3。`matrix()` 函数中还可以设定其他的函数，如利用 `ncol` 参数来设定矩阵的列数，`nrow()` 函数和 `ncol()` 函数能够查看矩阵的行数和列数。在 R 控制台中输入“`?matrix`”可以查看 `matrix()` 函数的帮助文档。

查看 `matx1`，容易发现这是一个3行4列的矩阵，序列 x 逐列填充了这个矩阵。由于12正好是3的倍数，因此序列 x 恰好填满了一个整齐的矩阵，倘若元素个数和矩阵行数不成倍数关系，R 就会自动用前几个元素填满矩阵的最后一列，并给出一个警告。

R 在返回矩阵时，同时还返回了行和列的下标。由于矩阵是一个二维变量，所以矩阵中的元素需要两个下标来标识位置。与向量元素相似，矩阵元素的下标同样用 `[]` 符号括起来，并用逗号分开两个不同下标。如 `[2,3]` 就表示处于矩阵第二行、第三列位置的元素。在 R 控制台中输入命令 `matx1[2,3]` 即可查看矩阵 `matx1` 中下标为 `[2,3]` 的元素，输入命令 `matx1[2,]` 即可查看矩阵 `matx1` 中第二行中所有的元素，这种规则同样适用于进行矩阵运算。

```
> rownames(matx1) <- c("x1","x2","x3")
> colnames(matx1) <- c("y1","y2","y3","y4")
> matx1
      y1 y2 y3 y4
x1    1  4  7 10
x2    2  5  8 11
x3    3  6  9 12
> as.vector(matx1)
[1]  1  2  3  4  5  6  7  8  9 10 11 12
```

显然，矩阵这种二维数据结构是最常用的结构，它就像一张大表，能够放入一行行个案和一系列变量。因此，为矩阵命名是非常必要的一件事，尤其是为列命名。在矩阵中为行命名的函数是 `rownames()` 函数，为列命名的函数是 `colnames()` 函数。这两个函数的用法都和 `names()` 函数类似，将一个字符向量中的元素逐个设为矩阵的行或列的名字。查看 `matx1` 中存储的内容，此时矩阵的名称都设定完毕，矩阵中的信息将更加丰富。

二维的矩阵同样可以压缩为一维的向量，上述代码中最后一行代码使用 `as.vector()` 函数完成了这项任务。由于我们在后续程序中并不打算再次使用这个向量，因此这行代码并没有使用“`<-`”符号将它赋给某个变量，于是 R 直接返回了这个向量中的内容。此时矩阵 `matx1` 再次成为一个1~12的序列。

```

> class(matx1)
[1] "matrix"
> cbind(matx1,matx1)
      y1 y2 y3 y4 y1 y2 y3 y4
x1  1  4  7 10  1  4  7 10
x2  2  5  8 11  2  5  8 11
x3  3  6  9 12  3  6  9 12
> rbind(matx1,matx1)
      y1 y2 y3 y4
x1  1  4  7 10
x2  2  5  8 11
x3  3  6  9 12
x1  1  4  7 10
x2  2  5  8 11
x3  3  6  9 12

```

除 `mode()` 函数能够查看的元素类型外，变量的类型同样很重要。上述代码中第一行代码使用 `class()` 函数查看了 `matx1` 的变量类型，R 的返回结果为矩阵。`class()` 函数能够查看多种变量的类型，与元素类型相似，只有在变量类型已知时我们才能正确地选择操作函数。

合并矩阵时需要专门的函数。`cbind()` 函数用于按列合并矩阵，`rbind()` 函数用于按行合并矩阵。上述代码分别使用这两个函数将两个 `matx1` 矩阵合并为一个新矩阵，在按列合并时，新矩阵的列数将增多，按行合并时，新矩阵的行数将增多。`cbind()` 函数和 `rbind()` 函数同样可以拼接两个不同的矩阵，不过在使用 `cbind()` 函数时，被拼接的矩阵的行数务必一致；在使用 `rbind()` 函数时，被拼接的矩阵的列数务必一致，否则 R 将报错。

```

> matx2 <- matrix(c(13:24),nrow=4)
> colnames(matx2) <- c("y5","y6","y7")
> rownames(matx2) <- c("x4","x5","x6","x7")
> matx1%*%matx2
      y5  y6  y7
x1 430 452 474
x2 500 526 552
x3 570 600 630

```

在行列数一致时，两个矩阵同样能够完成四则运算，还有一些特别的矩阵运算也很实用。上述代码首先创建了一个 4 行 3 列的新矩阵 `matx2`，在创建 `matx2` 时我们用一种更简洁的方式写出了 `matrix()` 函数。第 4 行代码使用 “`%*%`” 运算符计算了 `matx1` 和 `matx2` 的内积，内积结果中分别保留了 `matx1` 中的行名称和 `matx2` 中的列名称。

```

> matx3 <- matrix(c(3,0,3,2,5,7,1,-3,5),nrow=3)
> t(matx3)
      [,1] [,2] [,3]
[1,]    3    0    3
[2,]    2    5    7
[3,]    1   -3    5

```

```
> solve(matx3)
      [,1]      [,2]      [,3]
[1,] 0.43809524 -0.02857143 -0.10476190
[2,] -0.08571429 0.11428571 0.08571429
[3,] -0.14285714 -0.14285714 0.14285714
```

上述代码创建了一个行数和列数相同的方阵。方阵是最特殊的一种矩阵，它能够进行的运算也最丰富。`t()` 函数的结果是矩阵的转置，`solve()` 函数的结果则是矩阵的逆。能够用于方阵的函数还有用于进行奇异值分解的 `svd()` 函数等。

1.4.3 数组中的维度函数

与其他语言中定义的数组相类似，R 中的数组是一种高维的数据结构。矩阵可以看作数组的一种特殊形式，由于维数过高时无论作图还是运算都非常不方便，因此数组是最不常用的一种变量。

```
> ary1 <- array(1:12,dim=c(2,3,2))
> ary1
, , 1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2
      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
> class(ary1)
[1] "array"
```

上述代码使用 `array()` 函数创建了数组 `ary1`。`array()` 函数中第一个参数表示使用从 1 到 12 的序列创建矩阵，第二个参数 `dim` 则指明数组共有 3 个维度，维度值分别是 2、3、2。

查看 `ary1` 中存放的数据，它由两个二行三列的矩阵构成，每一个矩阵又有一个属于自己的序号。观察 `[]` 框中序号和逗号之间的顺序，可以发现，`ary1` 中第一维由两个矩阵的行构成，第二维由两个矩阵的列构成，第三维则由矩阵前方的序号构成（这两个序号前面都有两个逗号，表示它们是第三维）。

```
> dimnames(ary1) <- list(
+ c("a", "b"),
+ c("c", "d", "e"),
+ c("f", "g"))
> ary1
, , f
```

```

      c d e
a 1 3 5
b 2 4 6

, , g

      c d e
a 7 9 11
b 8 10 12

```

由于数组的维数是不固定的，因此为数组命名时统一由 `dimnames()` 函数完成。上述代码中首次出现了代码连接符。为了整洁美观，我们把较长的第一句代码拆成4行短代码。第一行代码结束时 R 检测到这句代码还未结束，便自动生成一个加号用以连接下一行代码。这种形式在复杂程序中经常出现。

列表函数 `list()` 将三个 `c()` 函数生成的向量组合起来，并用来创建 `ary1` 中的名称，其中第 1 个向量中的元素用于为 `ary1` 的第一维变量命名，第 2 个向量中的元素用于为第二维向量命名，第 3 个向量则和第三维形成对应关系。

```

> ary2 <- array(1:6,dim=c(2,3))
> ary2
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> class(ary1);class(ary2)
[1] "array"
[1] "matrix"

```

在创建数组并命名的过程中我们已经发现，维数多于二维的数组表现起来较为烦琐，每个维度的意义难以理解，运算起来更是非常不方便，远不如多创建几个矩阵方便。作为数组的一种特殊形式，矩阵同样可以通过 `array()` 函数创建。上述代码中 `array()` 函数的 `dim` 参数仅设置了两个维度，此时数组 `ary2` 看起来与矩阵似乎毫无区别，为了确定二维数组和矩阵是否一致，第三行代码中利用 `class()` 函数分别查看了 `ary1` 和 `ary2` 的变量类型，`ary1` 仍是数组类型，而 `ary2` 则变为矩阵类型。

1.5 R 的列表和数据框

除向量、矩阵和数组外，列表和数据框也是 R 中两类基本的数据结构，这二者比向量、矩阵和数组更加灵活，是应用范围更广的两类变量。本节简单介绍列表和数据框的特性，并将它们与向量、矩阵和数组进行对比。

1.5.1 列表的特性和编辑方法

列表是向量的一种特殊形式，它同时又与数组的形式有些相似。在 1.3 节中我们为

数组 `ary1` 命名时创建了第一个列表，其中包含三个长度分别为 2、3、2 的元素。

```
> lst <- list(c(1,2),c(3,4,5),c(6,7))
> lst
[[1]]
[1] 1 2

[[2]]
[1] 3 4 5

[[3]]
[1] 6 7

> names(lst) <- c("one","two","three")
> lst
$one
[1] 1 2

$two
[1] 3 4 5

$three
[1] 6 7
```

上述代码中第一行代码再次创建了一个包含三个长度分别为 2、3、2 的元素的列表 `lst`，`list()` 函数中使用逗号将三个元素分开，每个元素又由 `c()` 函数构建。不妨将列表 `lst` 看作一个包含三个元素的向量，只不过列表中每个元素又由更小的元素单位组成。查看 `lst` 中存放的数据，其存放数据的方式与数组有些类似，不过列表只具有两个维度，每个维度的下标标识也与数组不相同。

列表的命名函数与向量一致。使用 `names()` 函数可以很方便地为列表的每个元素命名，再次查看 `lst`，其中的元素已经被命名完毕，注意元素名称由一个 `$` 符号标出。

```
> lst$one <- c("a","b")
> lst
$one
[1] "a" "b"

$two
[1] 3 4 5

$three
[1] 6 7

> lst$four <- list(c(8,8),c(9,9))
> lst
$one
[1] "a" "b"
```

```

$two
[1] 3 4 5

$three
[1] 6 7

$four
$four[[1]]
[1] 8 8

$four[[2]]
[1] 9 9

> unlist(lst)
one1 one2 two1 two2 two3 three1 three2 four1 four2 four3 four4
"a" "b" "3" "4" "5" "6" "7" "8" "8" "9" "9"

```

与仅能存放一种数据类型的向量不同，列表中能够存放多种类型的数据。上述代码中第 1、2 行代码将列表 `lst` 中的第 1 个元素替换为了字符型数据 `a`、`b`；第 4、5 行元素在列表 `lst` 中增加了一个新的元素 `four`，新元素是一个包含两个元素的小列表，在列表中嵌入列表是合法的做法。此时 `lst` 中的数据结构变得更加复杂，显然列表中嵌入列表这种做法的可读性并不强，因此通常不这样做。

在前 4 行代码中使用“列表名 + \$ + 元素名”这种形式来指定操作对象，这是列表操作中较为特殊的一点，除非事先用 `attach()` 函数绑定列表，否则不能直接使用元素名。

列表同样可以转换为向量形式。上述代码中第 5 行代码使用 `unlist()` 函数将 `lst` 转换为向量，向量的元素名由列表的元素名生成。由于 `lst` 中既有字符元素，也有数值元素，而数值型可变为字符型，反过来则不合法，因此转换后的向量是字符型的（这一点由向量元素被双引号引起可知）。使用 `as.list()` 函数也可以将向量转换为列表。

```

> lst2 <- c(list(zero=c(1,2)),lst)
> lst2
$zero
[1] 1 2

$one
[1] "a" "b"

$two
[1] 3 4 5

$three
[1] 6 7

$four
$four[[1]]

```



```
[1] 8 8

$four[[2]]
[1] 9 9
```

`c()` 函数既可以拼接向量，也可以拼接列表；既可以将列表和向量拼接起来，也可以将列表和列表拼接起来。上述代码拼接了两个小列表，其中第一个列表直接在 `c()` 函数内用 `list()` 函数生成，第二个列表则是 `lst` 列表。我们在生成第一个列表时直接在 `list()` 函数内命名，这种简洁的做法也是合法的。

```
> lst$three+lst2$zero
[1] 7 9
> lst[[3]]+lst2[[1]]
[1] 7 9
> lst[3]+lst2[1]
错误于lst[3] + lst2[1] : 二进制运算符中有非数值参数
```

在对列表进行运算时有两种方法。上述代码中第一行代码利用元素名进行运算，第2行代码利用元素下标进行运算，这两条代码相互等价。由于列表中元素的长度和类型不要求一致，因此对列表中元素进行运算时需要确认被运算的两个元素的长度和类型符合运算条件，否则会报错。

使用列表的下标时也需要格外小心。在第2行代码中使用两个方括号括起了下标数，这样取出的内容就是元素中的内容，当只用一个方括号括起下标时，取出的内容就是该元素。这是两个不同的概念，具体来说，第2行代码取出内容的类型是数值，而第3行代码取出内容的类型则是列表，因此第3行代码会报错，不过这并不是说“`lst[3]`”这种操作是不合法的，它只是不能够用于四则运算。

1.5.2 数据框的创建和基本操作

在所有的数据结构中，数据框是最常用的一类结构。它与矩阵相似，具有规整的二维结构，同时数据框允许在不同列中存储不同的数据类型，这使得数据框要比矩阵更实用。

```
> df <- data.frame(x=c("a","b","c","d"),y=c(1,2,3,4),z=c(5,6,7,8))
> df
  x y z
1 a 1 5
2 b 2 6
3 c 3 7
4 d 4 8
> df$z0 <- df$z>6
> df
  x y z    z0
1 a 1 5 FALSE
2 b 2 6 FALSE
```

```
3 c 3 7 TRUE
4 d 4 8 TRUE
```

上述代码显示了数据框的创建和增添。其中，第 1 行代码使用 `data.frame()` 函数创建了一个 4 行 3 列的数据框 `df`，在 `data.frame()` 函数中也规定了每一列的名称。在创建数据框时如果不指定名称，R 会自动为每一列分配名称，通常是 `X1`、`X2`、`X3`……这种形式。

与列表相似，在没有使用 `attach()` 函数绑定数据框之前，调用数据框中的变量时需要使用“数据框名+\$+变量名”的格式。上述代码中第三行代码在数据框 `df` 中添加了一个新列 `z0`，`z0` 中的元素是判断 `df` 中 `z` 变量是否大于 6 的逻辑值，若不等式成立，`z0` 的结果就是“TRUE”，反之则是“FALSE”。

```
> df[2:3,-4]
  x y z
2 b 2 6
3 c 3 7
> df[df$y>3,c("x","z")]
  x z
4 d 8
```

数据框中元素下标的表示方法与矩阵类似，在数据框中，使用元素下标进行操作非常方便。上述代码利用下标筛选查看了 `df` 中的数据，其中，第 1 行代码查看了第 2~3 行的数据，参数“-4”表明返回结果中去掉第 4 列（但 `df` 中的数据并未改变）；第 2 行代码筛选出数据框 `df` 中 `y` 变量的值大于 3 的数据，并查看了 `x` 列和 `z` 列，注意方括号中第一个参数需要写成“数据框名+\$+变量名”格式，否则 R 将报错。

```
> nrow(df)
[1] 4
> names(df)[4] <- "z1"
> df
  x y z   z1
1 a 1 5 FALSE
2 b 2 6 FALSE
3 c 3 7  TRUE
4 d 4 8  TRUE
```

数据框的拼接与矩阵类似，`cbind()` 函数和 `rbind()` 函数能够将两个数据框按列或按行合并，不过被合并的数据框需要满足列数相等或行数相等。此外，由于实际应用中数据框往往比较大，因此统计数据框的行数和列数是非常必要的，上述代码中第 1 行代码使用 `nrow()` 函数统计了数据框的行数，与之相对的是 `ncol()` 函数，它可以统计数据框的列数。

上述代码中第 2 行代码修改了 `df` 中第 4 列的列名。数据框的列名称总是被存储在一条向量中，因此 `df[4]` 这种写法就相当于 `df[,4]`，表示 `df` 中的第 4 列，相对的，`df[,4]` 则表示数据框 `df` 中的第 4 行。

```
> lst <- as.list(df)
```

```

> lst
$x
[1] a b c d
Levels: a b c d

$y
[1] 1 2 3 4

$z
[1] 5 6 7 8

$z1
[1] FALSE FALSE TRUE TRUE

> df2 <- as.data.frame(lst)
> df2
  x y z  z1
1 a 1 5 FALSE
2 b 2 6 FALSE
3 c 3 7  TRUE
4 d 4 8  TRUE

```

数据框和列表之间也存在相互转换的函数。其中，`as.list()` 函数能将数据框转换为列表，`as.data.frame()` 函数能将列表转换为数据框。查看第 1、2 行代码得到的返回结果，`df` 中的 4 列被存储为 4 个元素，这 4 个元素顺序构成了列表，需要注意的是，其中元素 `x` 有一个 `Levels` 标注，这表明元素 `x` 是因子型的，这是由于数据框会自动将字符型变量存储为因子型，`stringAsFactors()` 函数能够避免这种强制转换。

上述代码中第 3、4 行代码显示了列表如何转换为矩阵，并不是所有的列表都可以转换为矩阵，只有元素长度都相等的列表才可以。除列表外，矩阵和一些其他形式的数据也可以转换为数据框。

1.6 R 数据文件的载入和载出

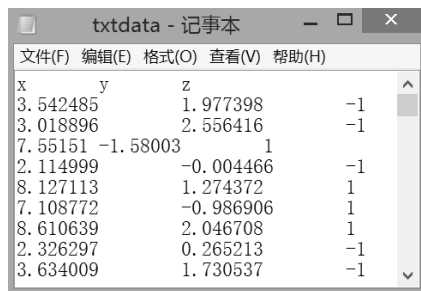
R 支持多种数据文件的载入和载出。本节讨论纯文本文件、非结构化文本文件、其他统计软件格式文件和网页文件等多种不同的文件类型，并介绍一些基本的文件读取函数的用法。

1.6.1 结构化纯文本文件的读取和输出

结构化纯文本文件是最常用到的一类文件，也是最好处理的一类文件。这类文件中每行都存储一条数据，每条数据中的元素又由分隔符分开，当分隔符是逗号时，纯文本文件就是 `csv` 格式的；当分隔符是空格时，纯文本文件就是 `txt` 格式的。

图 1.4 显示了文件 `txtdata` 中的前几条数据。显然，这个文件中有三列数据，列名称

分别为 x 、 y 、 z ，由于这个文件中的数据是由制表符分割的，因此该图中两个数据之间的缝隙显然比较大，对于 R 来说，制表符和空格造成的效果是一样的，因此这并不影响代码的执行。



x	y	z
3.542485	1.977398	-1
3.018896	2.556416	-1
7.55151	-1.58003	1
2.114999	-0.004466	-1
8.127113	1.274372	1
7.108772	-0.986906	1
8.610639	2.046708	1
2.326297	0.265213	-1
3.634009	1.730537	-1

图 1.4 txtdata 文件数据格式

```
> getwd()
[1] "C:/Users/Documents"
> txt <- read.table("txtdata.txt",header=TRUE)
> head(txt)
      x      y      z
1 3.542485 1.977398 -1
2 3.018896 2.556416 -1
3 7.551510 -1.580030  1
4 2.114999 -0.004466 -1
5 8.127113  1.274372  1
6 7.108772 -0.986906  1
```

在正式读取数据之前，还需要先将数据文件放到 R 的工作目录下。如果你不知道你计算机上 R 的工作目录是哪条路径，可以使用命令“getwd()”来查询，比如笔者的查询结果为“C:/Users/Documents”，这条路径就是笔者的 R 的工作目录。

将 txtdata 文件放到工作目录下，即可用 read.table() 函数来读取文本文件中的内容。read.table() 函数中第一个参数指定读取的文件是 txtdata.txt 文件，注意，这个参数中的文件扩展名绝对不能省略，如果没有写明文件扩展名，R 就会报错；第二个参数指定 header 为真，也就是读取的文件中存在表头，由于 txtdata 文件中写明了数据的列名称是 x 、 y 、 z ，因此需要设定这个参数。

第 3 行代码使用 head() 函数查看了 txt 中的前 6 行数据，如果直接查看 txt，显然，R 就会刷出好几屏的数据。从 head() 函数的返回结果可知，此时 txtdata 中的数据已经成功载入到 R 中，这三列数据规规矩矩地排列在一起，列名称也没有搞错。

如图 1.5 所示，文本文件 csvdata 中存放的数据和文件 txtdata 中的数据是一致的，只不过 csvdata 中的数据是由逗号进行的分割，而且这份数据没有表头。同样可以用 read.table() 函数读取数据。

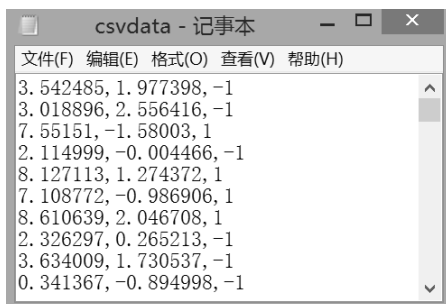


图 1.5 csvdata 文件数据格式

```
> csv <- read.table("data/csvdata.csv", sep=", ")
> head(csv)
      V1      V2 V3
1 3.542485 1.977398 -1
2 3.018896 2.556416 -1
3 7.551510 -1.580030  1
4 2.114999 -0.004466 -1
5 8.127113 1.274372  1
6 7.108772 -0.986906  1
```

R 可以读取并未放在工作目录下的文件，比如 csvdata 文件的存放目录为 "C:/Users/Documents/data"，在读取文件时，read.table() 中的第一个参数写明读取的是 data 文件夹下的 csvdata 文件（data 前的工作目录已被省略）。类似地，如果想要 R 读取存放在其他盘的文件，只需在 read.table() 中的第一个参数处写明文件目录即可。

read.table() 中的第二个参数设定为 sep 值为逗号，即文件的分隔符为逗号。通过设定 sep 值，read.table() 函数可以读取由不同分隔符分割的多种文件。除 header 和 sep 外，read.table() 函数还提供了用于指定读取行数的 nrow 参数、跳过开头几行的 skip 参数、确定行名称和列名称的参数、指定文件的字符编码参数等，在 R 控制台中输入 “?read.table” 可以查看它的更多信息。

与 read.table() 非常相似的函数还有分隔符默认为逗号的 read.csv() 函数；小数位默认为句号、分隔符默认为制表符的 read.delim() 函数等。此外还有一个更为灵活的 scan() 函数用于读取不标准的文本文件，它能够写出更复杂的读取命令。无论是哪个函数，它们都会自动将读取的数据设置为数据框格式。

```
> write.csv(csv, file="csv2.csv")
> write.table(txt, file="txt2.txt")
```

与读取文件相对应的是文件的输入，显然，write.csv() 函数用于输出 csv 格式的文本文件，write.table() 函数用于输出 txt 格式的文本文件，另外，还有一个 write() 函数也能实现同样的功能。上述两条代码利用数据框 csv 生成了 csv2.csv 文件，利用数据框 txt 生成了 txt2.txt 文件，这两个文件都存放在 R 的工作目录下，若要存放到其他目录下，只需在 file 参数中注明即可。

1.6.2 其他文件的读取和输出

文本文件中除存储用分隔符分开的数据表外，也可能存储着真正的字符数据。R 提供了 `readLines()` 函数来逐行读取文本信息，作为例子，`emaildata.txt` 是一个放在 R 工作目录下的邮件。

```
> email <- readLines("emaildata.txt")
> head(email)
[1] "What is going on there?"
[2] "I talked to John on email. We talked about some computer stuff that's it."
[3] ""
[4] "I went bike riding in the rain, it was not that cold."
[5] ""
[6] "We went to the museum in SF yesterday it was $3 to get in and they had"
```

上述代码使用 `readLines()` 函数读取了 `emaildata.txt` 中的信息，由于字符文件的结构松散，因此表头、分隔符等参数都是不需要设定的。`head()` 函数查看了 `email` 的前 6 个元素，它们正好和 `emaildata` 中的前 6 行相对应。显然，这种数据结构并不是数据预处理的终点，通常还需进一步处理后，非结构的文本文件才能用于数据分析。

```
> class(email)
[1] "character"
> writeLines(email, "email2.txt")
```

与 `readLines()` 函数相对应的是 `writeLines()` 函数，这个函数将字符向量逐行地输出为一个文本文件。实际上，`readLines()` 函数读入的数据也是向量类型的。需要注意的是在 `writeLines()` 函数中指定输出文件的参数写作 `"email2.txt"`，而 `file="email2.txt"` 这种写法是不合法的。

除结构化文本文件和非结构化文本文件外，来自其他统计软件的数据、来自网页的数据和来自数据库的数据也是常见的数据源，R 同样为这些数据提供了用于读取和输出的函数。

其他统计软件的数据主要包括来自 Excel、SAS、SPSS、Stata、MATLAB 等统计软件的数据。`xlsx` 包提供的 `read.xlsx()` 函数能够读取 Excel 软件所生成的 `xlsx` 文件；`foreign` 包同时提供了能够读取 SAS 文件的 `read.ssd()` 函数、能够读取 DTA 文件（由 Stata 软件生成）的 `read.dta()` 函数和能够读取 SPSS 文件的 `read.spss()` 函数；`R.matlab` 包则提供了能够读取 MATLAB 文件的 `readMat()` 函数。除这些方法外，还可以先将其他软件格式的数据转换为文本文件格式，再读取到 R 中。

来自网页的数据成分较为复杂，有一些网站提供了对应的 `csv` 数据文件以供下载和读取；另一些网站则提供了专门的包，包中具有读取数据的函数；其他那些既不提供下载文件也不提供包和函数的网站就显得较为复杂，简单来说，我们需要人工分析其网站源代码，找到有效数据，并找到数据在前后文中的标签，通过设置标签来使 R 下载数据，下载完毕后还需进一步加工才可以将数据正式投入数据分析中。即便 R 提供了众多函数，这也仍是一件很复杂的事情。

来自数据库的数据是最后一类常见的数据。数据库能够管理海量数据，并允许许多人访问，这在企业应用中最为常见。要从数据库中获取数据，就首先要加载使 R 能够连接到数据库的包，以 MySQL 为例，对应的包就是 RMySQL 包，设置好驱动器类型和文件路径后，即可从 MySQL 中读取文件。

1.7 向 R 中安装包

在 1.5 节关于数据读取的讨论中，多次提到了包的概念。在 R 中，包是一个重要的概念，R 社区内所有用户的共同努力使得如今已有成千上万的 R 包可供使用，这些 R 包拓展了 R 的用途，对数据分析师来说，如何选择合适的 R 包用以数据分析是一个重要的命题。

在下载 R 时，我们已经附带着下载了一些基本包，但这些包远远不够，比如，ggplot2 程序包就是一个非常重要、应用非常广泛的绘图包。向 R 中安装包时，利用 GUI 是最方便的选择。在正式开始安装包之前，用户务必要保证自己的电脑处于联网状态，并不受下载限制。

在 RGui 中的顶部菜单栏中有一个程序包下拉菜单，这一个菜单中的选项全都是关于 R 程序包的安装与维护的。在正式安装程序包之前，我们需要首先设定镜像。通常来说，无论是选择“设定 CRAN 镜像”，如图 1-6 所示，还是选择“安装程序包”，RGui 都会首先让用户设定镜像。



图 1.6 选择一个 CRAN 镜像

在 RGui 提供的镜像中，来自中国的镜像共有 4 个，（见图 1.7），分别是 China(Beijing1)、China(Beijing2)、China(Hefei) 和 China(Xiamen)。无论是本国的镜像还是外国的镜像，它们提供的东西都是完全一致的，因此建议你选择一个最近的镜像，这样能保证下载速度是最快的。

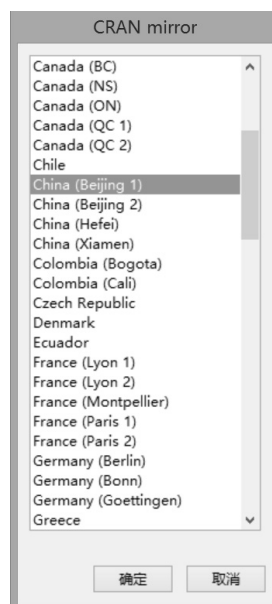


图 1.7 镜像设定对话框

在设定好镜像后，用户即可选择程序包进行安装。选择“安装程序包”，RGui 将提供一个相当长的列表，其中按照字母顺序列出了绝大部分已有的 R 包。从中找到 `ggplot2` 包，如图 1.8 所示，选中后单击“确定”按钮，RGui 将自动从 R 主页上下载 `ggplot2` 相关的文件，并完成安装。有些包的安装需要依赖另一些包，此时 R 控制台将给出提示信息，只需先安装好依赖包，再安装需要的包即可。

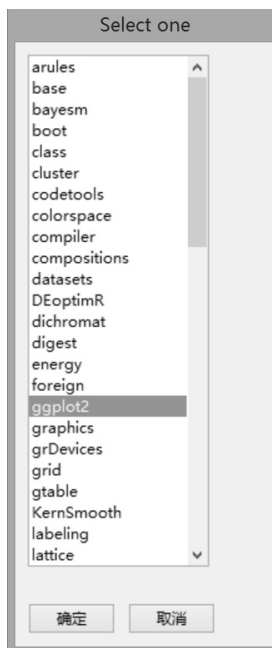


图 1.8 选择一个程序包进行安装

除利用 GUI 安装 R 包外，也可以直接使用编程的方式安装包。

```
> install.packages("ggplot2", repos="http://mirror.bjtu.edu.cn/cran/")
```

上述代码同样安装了 ggplot2 包，install.packages() 函数达到的效果和直接选择“安装程序包”一样。同时 install.packages() 函数设定的镜像网址是 <http://mirror.bjtu.edu.cn/cran/>，这是一个中国的网址，在 <https://cran.r-project.org/mirrors.html> 页面可以查看其他的中国镜像网址。

在 install.packages() 函数不起作用的时候，用户也可以直接进入 R 主页下载相关的 zip 文件，并进行本地安装程序包的操作。

安装好程序包后，在使用程序包中的函数时还需向 R 中加载包，此后才能调用包中的函数。

```
> library(ggplot2)
> help(package="ggplot2")
```

library() 函数用于向 R 中加载包，注意包的名称不需要引号。使用 help() 函数可以查看 ggplot2 中包含的全部函数及它们的说明文档。加载好包后，调用包中函数时直接写出函数名即可，只有在同时加载的多个包中出现了一样的函数时，才需要用“程序包名+::+函数名”的形式对多个名称一样的函数进行区分。

```
> update.packages()
> remove.packages("ggplot2")
```

上述两条代码是与包相关的最后两个重要代码，其中第 1 行代码对已有的包进行了更新，第 2 行代码则删除了包 ggplot2，当然，第 2 行代码也能够用于删除其他的包。

第 2 章 原始数据的探索与预处理

第 1 章详细地介绍了 R 的基础知识，第 2 章围绕数据的探索与预处理进一步展开。数据的探索和预处理是数据分析中的第一项任务，本章将讨论最常见的数据探索方法和预处理方法，数据框是本章讨论的主要形式，在第二部分的数据挖掘实例中对本章内容还将更深入地进行讲解。

2.1 度量数据集的集中程度

均值、分位数和众数是一列数据的基本特征，也是最常见的统计量，在对原始数据进行探索时，这三个量能够简单地刻画出数据的轮廓，度量出数据集的集中程度。R 包 `datasets` 中存储了许多示例数据集，其中 `ChickWeight` 数据集是一份关于销售数据的领先指标数据。

```
> library(datasets)
> head(BJsales)
[1] 200.1 199.5 199.4 198.9 199.0 200.2
> class(BJsales)
[1] "ts"
> mode(BJsales)
[1] "numeric"
> length(BJsales)
[1] 150
```

向 R 中载入程序包 `datasets` 后，数据集 `BJsales` 即可直接调用。上述代码分别查看了 `BJsales` 的一些基本性质，其中，第 2 行代码查看了 `BJsales` 的前 6 个元素，显然，`BJsales` 并不是一个数据框。第 3 行代码查看了 `BJsales` 的属性，返回结果为 `"ts"`，这表明 `BJsales` 中存放的是一列时间序列向量，即 `BJsales` 中的数据是按照时间顺序排列的。在控制台中输入 `"?ts"` 可以查看更多关于数据类型 `ts` 的信息。

`mode()` 函数的返回结果表明 `BJsales` 中的数据是数值型的，因此，计算 `BJsales` 的均值、中位数和分位数是有意义的。`length()` 函数给出了 `BJsales` 中元素的个数，由返回结果可知，`BJsales` 中共有 150 个元素。

```
> mean(BJsales)
[1] 229.978
> mean(BJsales,trim=0.1)
[1] 229.715
> median(BJsales)
[1] 220.65
```

均值和中位数是有些相似的两个统计量。均值是用所有元素的和除以所有元素的个

数得出的结果，其计算公式为 $\frac{\sum_{i=1}^n x_i}{n}$ 。中位数则是将元素按顺序排列后处于正中间的数，

当对偶数个元素计算中位数时，中位数是最中间两个数的均值。显然，均值和中位数都居于总体数据的中央位置，但均值易受异常值影响，而中位数则不。

上述代码中前两行代码都使用了 `mean()` 函数，第1行代码计算了 `BJsales` 中全体元素的均值，第2行代码中的 `trim` 参数指定在计算均值时数值最大的10%的元素和数值最小的10%的元素都要去掉，`trim` 函数能够削弱异常值的影响。这两种计算方法的结果十分接近，说明 `BJsales` 中并无偏离大部分数据较远的异常值。

`median()` 函数给出了 `BJsales` 的中位数，中位数要比均值小一些，也就是说，`BJsales` 中较小的数据分布较集中，较大的数据分布较分散，数据总体呈右偏分布。

```
> quantile(BJsales)
 0%      25%      50%      75%     100%
198.600 212.575 220.650 254.675 263.300
> quantile(BJsales,c(0.85,0.9,0.95))
 85%      90%      95%
257.60 259.06 261.21
```

`quantile()` 函数能够给出数据的分位数，它有好几种用法，上述代码给出的是最常用的两种用法。其中，第1行代码直接给出了 `BJsales` 的五分位数，比较分位数之间的距离，容易发现上四分位数与中位数的距离要小于它与最小值的距离，下四分位数与中位数的距离要大于它与最大值的距离，而最小值与中位数的距离又小于最大值与中位数的距离，这也佐证了 `BJsales` 是右偏分布的结论。

第2行代码查看了 `BJsales` 中处于85%、90%和95%这三个位置的元素，通过设定 `c()` 函数给出的分位数位置，`quantile()` 函数可以查看处于任意百分位处的元素。显然，中位数是分位数的一个特例。

众数是一个较为特殊的统计量，它代表数据集中出现次数最多的数。将数据集中的数据按照出现次数的多少顺序排列后，我们能够大致地看出数据集的分散程度。然而由于 `R` 中浮点数的小数位很长，因此，统计每个值出现的次数并找出众数是不太可能的。`R` 中并没有计算众数的函数，不过在一些绘图函数中，众数被可视化地表达了出来。

2.2 度量数据集的分散程度

数据集的分散程度主要从方差、标准差、标准误、偏度系数和峰度系数等统计量加以考量。本节仍旧使用了领先指标销售数据的数据集，其分析结果将和2.1节的结果互为佐证。

2.2.1 极值、方差和标准差

极值、方差和标准差是较为常见的统计量，它们经常用于度量数据集的分散程度。

R 中内置了好几种函数用以计算这些统计量。

```
> range(BJsales)
[1] 198.6 263.3
```

极值是对数据集的极小值和极大值的总称。在第 1 章中已经提到过 `min()` 函数和 `max()` 函数，它们分别给出数据集的极小值和极大值，与之类似地是 `range()` 函数，它同时给出两个极值。上述代码是对 `BJsales` 应用 `range()` 函数的结果。另外，2.1 节中用于计算分位数的 `quantile()` 函数也能够用于计算极值。总的来说，`range()` 函数是计算极值时的首选。

极值具有容易理解的特点，但它也容易受异常值的影响。极值只能度量出数据集的两端范围，刻画不出其他那些非极值点的密集程度和分布情况。因此，极值提供的信息非常有限，它通常扮演着一个辅助分析的角色。

```
> var(BJsales)
[1] 461.3769
> sd(BJsales)
[1] 21.47969
```

方差是由统计学家规定的专门用于度量数据集分散程度的统计量，它的计算公式是

$$\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}$$

其中 \bar{x} 代表数据集的均值， n 则是数据集中数据的个数，在样本数据中为了修正误差，分母也会写成 $n-1$ ，而非 n 。从计算公式上看，方差计算的是数据集中全部数据与均值的差方的和除以数据个数的值，这个公式同时受到全体数据的影响，方差越大，则说明数据集中的数据离均值越远，数据越分散。

`var()` 函数能够计算数据集的方差（它默认的分母是 $n-1$ ），由上述代码可知，`BJsales` 的方差是 461.376 9，这是一个非常大的数，实际上它远远超过了 `BJsales` 的数据范围。为了使度量分散程度的统计量更加易读，对方差进行开方即可得到数据集的标准差。`sd()` 函数直接计算出了 `BJsales` 的标准差，显然，它和方差具有一个平方关系。标准差要比方差小得多，比较标准差和数据集范围，能够较容易地看出数据的分散程度。

`var()` 函数能够计算数据集的方差（它默认的分母是 $n-1$ ），由上述代码可知，`BJsales` 的方差是 461.376 9，这是一个非常大的数，实际上它远远超过了 `BJsales` 的数据范围。为了使度量分散程度的统计量更加易读，对方差进行开方即可得到数据集的标准差。`sd()` 函数直接计算出了 `BJsales` 的标准差，显然，它和方差具有一个平方关系。标准差要比方差小得多，比较标准差和数据集范围，能够较容易地看出数据的分散程度。

```
> c(mean(BJsales)-sd(BJsales),mean(BJsales)+sd(BJsales))
[1] 208.4983 251.4577
```

上述代码计算了 `BJsales` 中均值和方差的差值与和值，这两个数值恰好落在 `BJsales` 的极值之间，实际上无论哪个数据集，均值和方差的差值与和值总是落在极值之间。比较差值、和值与两个极值的差，如果差较大，数据就较集中；如果差较小，数据就较分散，观察差值、和值与数据集的两个四分位数是否相似也能得出同样的结果。

除与数据集的其他内部数据进行比较外，方差和标准差也能和其他数据集的方差和标准差进行比较。需要注意的是，方差和标准差的大小受量纲的影响，如果数据集中的数据本来就比较大，那么方差和标准差也会比较大；如果数据集中的数据本来就比较小，那么方差和标准差也会比较小。因此不同量纲之间的方差和标准差是不能直接相比的，在比较前需要先对数据集进行标准化。

2.2.2 标准误和偏度系数、峰度系数

除极值、方差和标准差外，还有一些其他的统计量也可以用于度量数据的分散程度。R 中并未为这些统计量一一编写现成的函数，有些统计量在使用时需要我们自己编写公式进行计算，比如标准误、偏度系数、峰度系数就是三个需要用户自己编写公式的实用统计量。

```
> error <- sqrt(var(BJsales)/length(BJsales))
> error
[1] 1.753809
```

上述代码计算了 BJsales 的标准误。标准误的计算公式为 $\sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n(n-1)}}$ ，很明显，

其中， $\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}$ 就是 var() 函数的计算公式。因此只需计算 var() 函数值与 BJsales 中数

据个数的差的开方，即可得到标准误。上述代码中第 1 行代码使用 length() 函数计算了 BJsales 中数据的个数，并用 sqrt() 函数实现了开方，由返回结果可知，BJsales 的标准误为 1.753 809。

标准误和标准差是两个不同的概念，标准误的计算公式可简化为 $\frac{s}{\sqrt{n}}$ ，这与标准差的公式显然不一样，标准误在标准差的基础上消去了数据量带来的影响，对数据量相差较大的多个数据集来说，标准误更具有比较意义。

```
> skewness<-length(BJsales)*sum((BJsales-mean(BJsales))^3)/
+ ((length(BJsales)-1)*(length(BJsales)-2)*sd(BJsales)^3)
> skewness
[1] 0.2858044
```

偏度系数和峰度系数是两个衡量数据集的分布形状的系数。偏度系数的计算公式为 $\frac{n \cdot \sum_{i=1}^n (x_i - \bar{x})^3}{(n-1)(n-2)s^3}$ ，其中，s 代表数据集的标准差。这是一个很复杂的公式，上述代码使用 length() 函数、sum() 函数、mean() 函数、sd() 函数等多个函数完成了复杂的计算过程，R 的返回结果显示 BJsales 的偏度系数为 0.285 804 4。

偏度系数是一个取值通常在 -3~+3 之间的值，它衡量了数据集的对称程度。数据越对称，偏度系数就越接近 0；数据越不对称，偏度系数就越远离 0。BJsales 的偏度系数是正的，这表明 BJsales 的右侧数据更分散，否则，它的偏度系数将是负的。

```
> m<-mean(BJsales)
> n<-length(BJsales)
> s<-sd(BJsales)
```

```
> kurtosis<-(n*(n+1)*sum((BJsales-m)^4)/((n-1)*(n-2)*(n-3)*s^4)-(
3*(n-1)^2/((n-2)*(n-3)))
> kurtosis
[1] -1.526321
```

峰度系数的计算公式更为复杂， $\frac{n(n-1)\sum_{i=1}^n(x_i-\bar{x})^4}{(n-1)(n-2)(n-3)s^4}-\frac{3\times(n-1)^2}{(n-2)(n-3)}$ 是它的具体表达形式，

为了减少编程量，上述代码中首先设置了三个辅助变量，而后使用这三个辅助变量进行了计算。显然，这种编程方式的易读性更强，写起来也更方便。

由 R 的返回结果可知，BJsales 的峰度系数为 -1.526 321。峰度系数由数据集的四阶矩计算得到，正态分布的峰度系数为 3，不过上述代码的计算公式中已减去 3，因此，只需将峰度系数与 0 进行比较即可得知数据集的分布峰度与正态分布的相似程度。峰度系数越接近 0，数据集的分布峰度就与正态分布越相似；峰度系数越远离 0，数据集的分布峰度就与正态分布越不相似。BJsales 的峰度系数为负，这表明 BJsales 中的数据较为集中，两侧数据较少；否则，BJsales 的峰度系数将为正。

正态分布具有良好的性质，因此我们总是希望数据集的分布尽可能地接近正态分布。偏度系数和峰度系数就是这样两个实用的函数，显然，当偏度系数和峰度系数都为 0 时，数据集就服从一个标准的正态分布。

除标准误、偏度系数、峰度系数外，还有极差、变异系数、样本校正平方和等许多有用的统计量，其中的大部分统计量 R 都没有为它们提供单独的函数，我们在使用这些统计量时可以自己编写函数，或者安装 R 中附带了这些函数的包。另外，有些数值摘要表中提供了这些统计量，也可以从数值摘要表中查询统计量的值。

2.3 创建一个数值摘要表

之前两个小节介绍了好几种在探索数据时常用的统计量，这些统计量共同刻画出数据的大致轮廓，自然的，如果有一个函数能够将这些统计量汇总起来，这将创造极大的方便。实际上，这种函数已经存在于 R 中，它们能够创建包含多个统计量的数值摘要表，为了展示数据汇总函数在数据框上的效果，本节使用 `attenu` 数据集作为示例。

```
> library(datasets)
> head(attenu)
  event mag station dist accel
1     1  7.0     117   12 0.359
2     2  7.4    1083  148 0.014
3     2  7.4    1095   42 0.196
4     2  7.4     283   85 0.135
5     2  7.4     135  107 0.062
6     2  7.4     475  109 0.054
```

在 R 中加载 `dataset` 包后，`attenu` 数据集即可直接调用。`attenu` 是一个与 Joyner-

Boore 衰变系数有关的数据集，使用 `head()` 函数查看 `attenu` 的前 6 行数据，由 R 的返回结果可知 `attenu` 中有 5 个变量，其中，`event` 存储了事件标签信息，`mag` 存储了镁信息，`station` 存储了观察站点信息，`dist` 存储了距离信息，`accel` 存储的则是时间调整信息。显然，它们都是数值型的。

```
> fivenum(attenu$dist)
[1] 0.5 11.1 23.4 47.7 370.0
> summary(attenu)
      event      mag      station      dist
Min.   : 1.00  Min.   :5.000  117    : 5  Min.   : 0.50
1st Qu.: 9.00  1st Qu.:5.300  1028   : 4  1st Qu.: 11.32
Median :18.00  Median :6.100  113    : 4  Median : 23.40
Mean   :14.74  Mean   :6.084  112    : 3  Mean   : 45.60
3rd Qu.:20.00  3rd Qu.:6.600  135    : 3  3rd Qu.: 47.55
Max.   :23.00  Max.   :7.700  (Other):147  Max.   :370.00
                        NA's    : 16

      accel
Min.   :0.00300
1st Qu.:0.04425
Median :0.11300
Mean   :0.15422
3rd Qu.:0.21925
Max.   :0.81000
```

R 提供了两个基础的数值摘要表函数：`fivesum()` 函数和 `summary()` 函数。其中 `fivenum()` 函数只作用于向量，`summary()` 函数既可以作用于向量，也可以作用于数据框。上述代码对 `attenu` 中的 `dist` 向量应用了 `fivenum()` 函数，对 `attenu` 应用了 `summary()` 函数。`fivenum()` 函数返回了向量的两个极值、两个四分位数和均值。该函数能够实现与 `quantile()` 函数相同的效果，但 `fivenum()` 函数的执行速度要快一些。`summary()` 函数的返回结果在 `fivenum()` 函数的基础上增加了一个中位数统计量。

R 控制台返回了 `event`、`mag`、`dist` 和 `accel` 变量的 6 个统计量，在 `station` 变量下则返回了一些其他数值。这是由于 `station` 是一个因子变量，因此 `summary()` 函数返回了 `station` 中出现次数最多的 5 个因子，并注明了其他因子总共出现了多少次，以及空缺值的个数。在 R 控制台中输入 `class(attenu$station)` 可以查看 `station` 的类型，输入 `?factor` 可以获取有关因子类型的更多信息。

```
> alldata <- function(x) {
+ var <- var(x)
+ sd <- sd(x)
+ med <- median(x)
+ R <- max(x) - min(x)
+ error <- sqrt(var/length(x))
+ m <- mean(x)
+ n <- length(x)
+ skewness <- n*sum((x-m)^3)/((n-1)*(n-2)*sd^3)
```

```

+ kurtosis <- (n*(n+1)*sum((x-m)^4))/((n-1)*(n-2)*(n-3)*sd^4)-(3*(n-1)^2/
((n-2)*(n-3)))
+ data.frame(var,sd,med,R,error,skewness,kurtosis)
+ }
> alldata(attenu$dist)
      var      sd  med      R      error skewness kurtosis
1 3865.117 62.17006 23.4 369.5 4.608352 2.909042  9.72888

```

除使用 R 中现有的函数外，我们也可以手动编写一个函数来创建数值摘要表。编写函数的格式为“函数名+<-+function(){}”，其中，function(){} 中的小括号内写入形参，花括号内写入函数语句块。上述代码块中第一条代码创建了一个名为 alldata 的函数，该函数只有一个形参 x，函数块中分别创建了 x 的方差、标准差、中位数、极差、标准误、偏度系数和峰度系数，并将这些统计量组合成一个数据框。由于数据框并未赋给某个变量，因此，在执行这条代码时控制台将直接返回该数据框。

上述代码的第二条代码调用了 alldata 函数，并将 attenu 中的 dist 变量作为参数传入了这个函数。R 返回了 dist 的数值摘要信息，显然，这份数值摘要表提供的信息要比 fivenum() 函数和 summary() 函数所能提供的丰富一些。

2.4 异常值的观测与说明

异常值是数据集中较为特殊的一类值，指距离大部分数据点明显较远的值。异常值的产生原因可能是数据录入错误、数据产生条件与其他数据不一致或出现小概率事件。异常值对数据分析结果会产生较大的影响。本节将讨论观测异常值的几种常见方法，以及异常值的处理方法。

2.4.1 利用箱线图观测异常值并处理

箱线图是一种利用极值、四分位数和中位数画出的图形，图中还标出了常规意义上的异常值，即距离中位数远于三个标准差的值。

```
> boxplot(attenu$dist)
```

上述代码使用 boxplot() 函数画出了 attenu 数据集中 dist 变量的箱线图。

图 2.1 所示为 dist 变量的箱线图显示了 R 的返回结果。由该图可知，箱线图由两条须线、一个箱子和一些异常值点组成，两条须线末尾的横线是 dist 的两个极值，箱子的上、下边缘是 dist 的分位数，箱子中的粗横线则是 dist 的均值。显然，boxplot() 函数在计算 dist 的极值、分位数和均值时，并没有把异常值考虑在内。

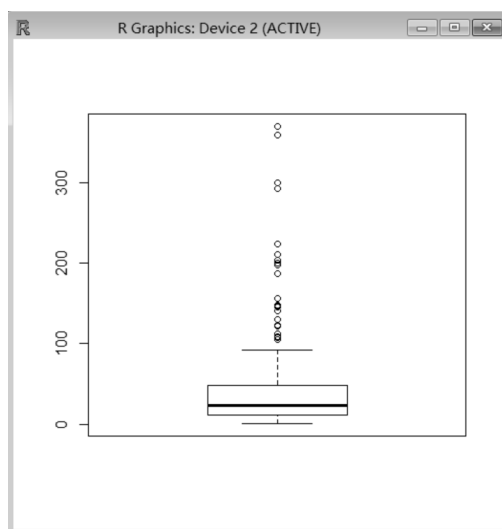


图 2.1 dist 变量的箱线图

观察箱线图，可以发现箱子和虚线都处于 0~100 之间，而该图中由圆圈标出的异常值点则处于 100~400 之间，这说明居于 `dist` 变量右侧的数据非常分散、稀疏。

```
> boxplot.stats(attenu$dist)
$stats
[1] 0.5 11.1 23.4 47.7 92.2

$n
[1] 182

$conf
[1] 19.1135 27.6865

$out
[1] 148 107 109 156 224 293 359 370 105 112 123 105 122 141 200 130 147 187
[19] 197 203 211 145 300
```

箱线图尽管直观易读，但其提供的信息并不精确，比如异常值究竟是哪些数。`boxplot.stats()` 函数提供了更详细的信息，上述代码查看了 `attenu$dist` 中更多的统计信息。R 控制台返回了 4 组信息，其中，`stats` 信息给出的是 `dist` 变量中去掉异常值后的极值、四分位数和中位数；`n` 给出的信息是 `dist` 中数据点的个数；`conf` 给出了 `dist` 的中位数 $\pm IQR/\sqrt{n}$ 的结果，它用于衡量中位数的一个置信区间；`out` 是最重要一个值，它给出了 `dist` 中异常值的具体数值。

```
> outlier <- which(attenu$dist %in% boxplot.stats(attenu$dist)$out)
> boxplot(attenu[-outlier,]$dist)
```

`boxplot.stats()` 函数中的 `out` 值为进一步处理异常值提供了便利，通常来讲，数据分析师在分析数据时会直接删掉异常值，上述代码实现了这个功能。其中，第 1 条代码

中的 `which()` 函数提供了一个判断条件，运算符 `%in%` 匹配了 `attenu$dist` 中与 `boxplot.stats(attenu$dist)$out` 有交集的部分，`which()` 函数将符合这些条件的数据赋给了 `outlier` 变量，此时 `outlier` 中存放了一系列数据序列，处于这些序列点处的点就是上文中找出的异常值。

上述代码中第 2 条代码画出了 `attenu$dist` 中去掉异常值后的箱线图，注意，“`attenu[-outlier,]$dist`”这种写法是合法的，如果写成“`attenu$dist[-outline,]`”形式，R 就会报错。

图 2.2 所示为去掉异常值后 `dist` 变量的箱线图，显示了 `boxplot()` 函数的返回结果，观察该图中的点，此时 `dist` 中的点已经限制在了 0~100 之间，那些处于 100~400 之间的异常值已经被去掉。但遗憾的是，`dist` 变量中再次出现了新的异常值。这些新的异常值处于 80~100 之间，仍旧集中在箱线图主体的上方，并且数量还不少。

这提示我们粗鲁地删除异常值是不太恰当的做法，对于本例中的 `dist` 变量来说，它右侧数据本来就比较分散，这种现象不一定是由于 `dist` 变量中的数据出现了误差或错误，也可能是因为 `dist` 变量本身就服从右偏分布。总的来说，异常值中蕴含着一些特殊的信息，处理异常值时需要结合背景条件和项目要求灵活处理，才能使数据分析结果最合理。

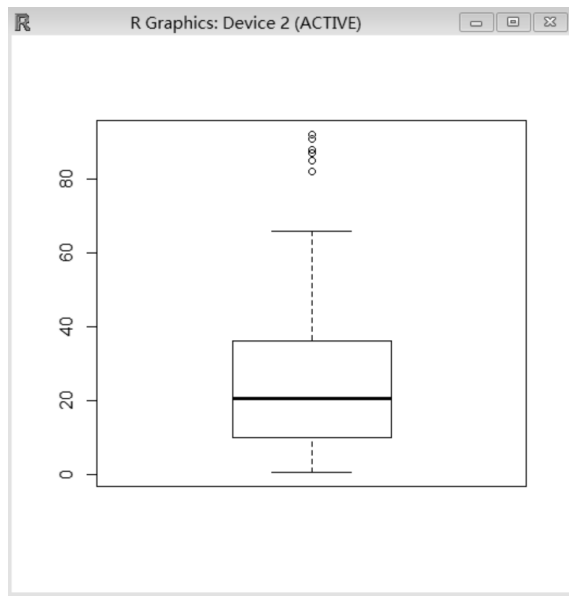


图 2.2 去掉异常值后 `dist` 变量的箱线图

2.4.2 异常值检测的其他情况和说明

将距离中位数远于三个标准差的数据点看作异常值的方法虽然简单流行，但它只能处理那些整体分布较为对称的数据集，整体分布有多个峰值时，这种判断方法将不起作用。

图 2.3 所示为具有三个峰的数据集，它的数据点主要集中在左侧，但右侧尾部又有两个小峰。对这种多峰分布的数据集，LOF 算法提供了用于计算局部异常因子的 `lofactor()` 函数。该函数能够找出那些离左边数据和右边数据都很远的数据点，并将这些

同时远离两个峰值的数据点视为异常值点，DMwR 包和 dprep 包均提供了它。

命令“`lofactor(attenu$dist,k=5)`”是 `lofactor()` 函数的一个简单应用，它规定 $k=5$ ，这将找出 `dist` 中最异常的 5 个数据点。

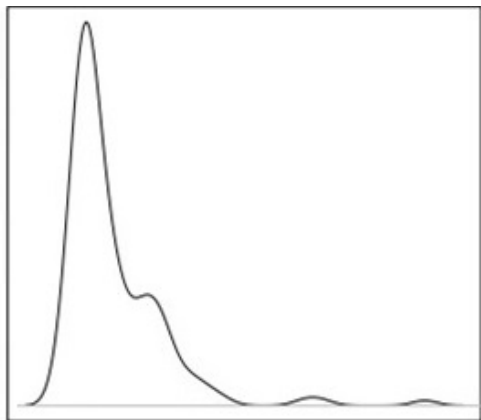


图 2.3 具有三个峰的数据集

时间序列数据是另一种特殊的多峰分布数据集，时间序列具有周期性，同时随着周期的增加，同一周期内的数据的波动程度也会发生改变，此时 `lofactor()` 函数不再起作用，`stl()` 函数能够利用稳健回归算法对时间序列进行分解，从而识别异常值，有关稳健回归的知识将在第 6 章中会有详细介绍。

除 LOF 算法外，`outliers` 包提供了更多的异常值检测算法，其中，Grubbs 方法和 Dixon 方法是较为经典的两种方法。Grubbs 方法将数据值和数据集均值的差的绝对值与数据集标准差相除，将除数和异常值分布表进行比较，除数大于临界值时，就认为该数据值是异常值，`Grubbs.test()` 函数具备利用 Grubbs 方法检验异常值的功能。Dixon 方法同样使用比较统计量和临界值的方法来检验异常值，`dixon.test()` 函数提供了多种统计量公式以供选择。

关于异常值，最后需要补充的一点是，本节所有关于异常值的讨论都围绕一元变量展开，在多元变量中探索异常值同样有其特殊的意义，比如同时在两个变量中都异常的异常值就格外具有探究意义。本节提到的函数同样可以应用于多元变量中异常值的探索工作，R 主页的帮助文档提供了大部分相关信息。

2.5 缺失值的填补与处理

缺失值是数据集中十分常见的一类数据，真实数据中往往存在或多或少的缺失值。缺失值处理得恰当与否直接影响分析结果的准确性。与异常值不同的是，如今缺失值的处理方法已经较为完善。本节将介绍最常用的几种处理缺失值的方法。

2.5.1 删除缺失值或对其进行简单填补

处理缺失值时，较为简单的方法是直接删除或使用均值、中位数等进行填补。本节从 `datasets` 包中选择了数据集 `airquality` 作为原始数据。这个数据集收集了一些关于纽约天气的数据。

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1   41    190   7.4   67     5   1
2   36    118   8.0   72     5   2
3   12    149  12.6   74     5   3
4   18    313  11.5   62     5   4
5   NA     NA  14.3   56     5   5
6   28     NA  14.9   66     5   6
```

向 R 中加载入 `datasets` 包后，利用 `head()` 函数即可查看 `airquality` 中的前 6 行数据。由 R 的返回结果可知，`airquality` 中存有 6 列变量，其中，第 1 列存放臭氧数据，第 2 列存放日光等级数据，第 3 列存放风力数据，第 4 列存放温度数据，第 5 列和第 6 列则分别存放了月份和日期的数据。显然，我们重点关心的是前 4 列数据。

```
> notna <- complete.cases(airquality)
> head(notna)
[1] TRUE TRUE TRUE TRUE FALSE FALSE
```

处理缺失值时，`complete.cases()` 函数能够找出数据框中不含缺失值的行。上述代码中第 1 行代码使用 `complete.cases()` 函数查询了 `airquality` 中的行，并将结果赋给 `notna` 变量。查看 `notna` 中的前 6 个元素，显然，`notna` 是一个向量元素，其中存放了逻辑向量，当元素为 `TRUE` 时，`airquality` 中对应的行就不含缺失值；当元素为 `FALSE` 时，`airquality` 中对应的行就含有缺失值。

```
> nrow(airquality)
[1] 153
> nrow(airquality[which(notna==FALSE),])
[1] 42
```

上述两条代码首先查询了 `airquality` 的行数，显然，`airquality` 中总共有 153 行数据。第 2 行代码查询了 `airquality` 中所有 `notna` 为 `FALSE` 的行的个数，显然，`airquality` 中含有缺失值的行共有 42 行。第 2 行代码较为复杂，`which()` 函数前面已经介绍过了，需要注意的是，“`notna==FALSE`”中双等号意味着判断双等号左、右两端的内容是否相等，只写一个等号时则代表赋值，在这里如果只写一个等号，R 就会报错。

```
> length(which(is.na(airquality$Ozone)==TRUE))
[1] 37
> length(which(is.na(airquality$Solar.R)==TRUE))
[1] 7
```

```
> length(which(is.na(airquality$Wind)==TRUE))
[1] 0
> length(which(is.na(airquality$Temp)==TRUE))
[1] 0
```

`airquality` 中包含缺失值的行的个数值得注意，每一列中缺失值的个数同样也值得注意。对于数据集 `airquality` 中的前 4 列数据来说，它们分别都可看作一个向量，因此，此时不能继续用 `nrow()` 函数来计数，`length()` 函数是较好的选择。

`is.na()` 函数同样返回一个逻辑向量，但与 `complete.cases()` 函数不同的地方在于 `is.na()` 函数仅能应用于向量，而 `complete.cases()` 函数同时可以应用于向量和数据框两种类型的数据；且 `is.na()` 函数返回 `TRUE` 时代表向量中该元素为空，返回 `FALSE` 时代表向量中该元素不为空，这一点与 `complete.cases()` 函数恰好相反。

使用 `which()` 函数同样可以写出与计算 `airquality` 中包含空缺值的行的个数类似的命令，显然，`Ozone` 中有 37 个空缺值，`Solar.R` 中有 7 个空缺值，而 `Wind` 和 `Temp` 中则没有空缺值。将 37 和 7 相加后并不等于 42，这是由于有些行的 `Ozone` 和 `Solar.R` 同时空缺，造成了这种差异。

```
> airquality <- na.omit(airquality)
> nrow(airquality)
[1] 111
```

处理缺失值时最简单的办法是，直接将含有缺失值的行删除。`na.omit()` 函数能简单地实现这一功能。在上述代码中我们删去了 `airquality` 中含有空缺值的行，查看 `airquality` 的行数，此时只有 111 行数据，这是 `airquality` 去掉 42 行含有空缺值的行后的结果。

尽管直接删去空缺值的方法粗暴、简单，但它也有适用的范围，以本例来说，总共只有 153 行数据，一下子删去 42 条数据未免删得太多，数据集中数据的个数显然和数据分析结果的准确度有很大关系，因此，只有空缺值的个数较少的时候才使用 `na.omit()` 函数，否则就使用填补空缺值的办法。

```
> airquality[is.na(airquality$Ozone),"Ozone"] <- mean(airquality$Ozone,na.rm=T)
> airquality[is.na(airquality$Solar.R),"Solar.R"] <- mean(airquality$Solar.R,na.rm=T)
```

上述两条代码利用列的均值替代了空缺值。由于被赋值对象中的 `is.na()` 函数是一个逻辑值，因此，这个赋值语句只有在 `is.na()` 的结果为真时才会执行，也就是仅对 `Ozone` 和 `Solar.R` 中的空缺值进行了替换。赋值语句使用了 `mean()` 函数计算 `Ozone` 和 `Solar.R` 的均值，由于这两列中都含有空缺值，因此，还需指定参数 `na.rm` 为真来确保计算均值时跳过空缺值，否则 `mean()` 函数的返回值将会是 `NA`。

需要补充说明的一点是，关于 `airquality` 的设置问题。在使用 `na.omit()` 函数删除 `airquality` 中含有空缺值的行时，我们已经修改了 `airquality` 中存放的数据，因此，想要使用 `mean()` 函数再次填补空缺值时必须重新载入 `airquality` 数据集，否则结果将会出错。在下文中使用其他方法填补空缺值时也同样需要重新载入数据集。

```
> airquality[is.na(airquality$Ozone),"Ozone"] <- median(airquality$Ozone,na.rm=T)
> airquality[is.na(airquality$Solar.R),"Solar.R"] <- median(airquality$Solar.R,na.rm=T)
```

类似地，上述两条代码使用 Ozone 和 Solar.R 的中位数替换了 Ozone 和 Solar.R 中的空缺值。在前文中已经提到过均值和中位数是非常相似的两个数，总的来说，当数据集分布较对称时，使用均值替换空缺值较好；当数据集分布较 Solar.R 不对称时，使用中位数替换空缺值较好。另外，也可以使用众数或其他能反映数据集中心的统计量来填补空缺值。

2.5.2 按照相关性对空缺值进行填补

使用均值或中位数来填补空缺值是一种较好理解、较好操作的方法，但它也是一种较为粗糙的方法。尤其是当数据集较大时，这种方法对数据集准确度的影响也会较大。按照相关性对空缺值进行填补是更加聪明的方法。

按照相关性填补空缺值同样分为如下两种方法：按照个案相关性或按照变量相关性。计算个案相关性并根据相关性填补空缺值较为简单，理解起来也较为容易。我们称数据表中的一行数据为一件个案，对数据集 airquality 来说，一件个案记录的就是一天的天气信息。计算个案相关性就是计算每一天和每一天之间的相似程度。

欧式距离是最常用于度量相似程度的统计量，欧氏距离的计算公式为 $d(x, y) = \sqrt{\sum_{i=1}^p (x_i - y_i)^2}$ ，其中， x_i 和 y_i 代表两个不同个案的不同分量，在 airquality 的例子中， p 就是除含空缺值的列外的 5 个列。比如在 airquality 的第 6 件个案中，Solar.R 为缺失值，则在计算第 6 件个案与其他个案的相似度时，计算的是其他个案除 Solar.R 值外的 5 个变量值与第 6 件个案的 5 个变量值的欧氏距离，显然，此时 p 为 5。

通过计算其他所有个案与第 6 件个案的欧氏距离可知，欧氏距离越小，该个案便与第 6 条个案越相似。找出与第 6 件个案最相似的 10 个个案，计算这 10 个个案 Solar.R 值的均值，该均值就是利用个案相似度为第 6 件个案计算出的 Solar.R 值。类似地，这种方法也可以用于填补其他个案空缺的 Solar.R 值。显然，这种方法的思想在于，既然两个个案的其他值都相似，那么这两个个案的 Solar.R 值也很可能相似。这种方法同样也可以用于填补空缺了其他值的个案。

尽管利用个案相似度填充空缺值的方法有些复杂，但 DMwR 包提供了一个非常好用的函数。

```
> library(DMwR)
> airquality <- knnImputation(airquality,k=10,meth="mean")
```

上述代码首先向 R 中载入了 DMwR 包，然后调用了 knnImputation() 函数，并为该函数指定了三个参数，其中，第一个参数指明 knnImputation() 函数应用的对象是 airquality 数据集；第二个参数指定用于计算统计量填补空缺值的相似个案个数为 10；第三个参数指定填补空缺值的统计量是均值。除这些参数外，knnImputation() 函数也提供了一些其他的参数选项，在 R 控制台输入 “?knnImputation” 可以查看更多的相关信息。

这种方法似乎既便捷又准确，但它同样存在明显的问题。以 `airquality` 为例，`knnImputation()` 函数进行计算时会将末两列的日期变量纳入欧氏距离计算公式中，这显然是不合理的。首先，日期变量是有序变量，并不适合欧氏距离公式；其次，日期变量与 `Ozone` 值或 `Solar.R` 值也不大可能有关系。因此使用 `knnImputation()` 函数填充空缺值也未必合理。

利用变量之间的相似度填补空缺值与利用个案相似度的思想是一致的。但利用变量相似度要更加复杂。首先，要查看不同列之间的相关系数；其次，挑选出那些与含有空缺值的列最相关的列；再次，要将含有空缺值的列看作因变量，将其他与之相关的列看作自变量，构建回归方程；最后，根据回归方程对空缺值进行预测和填补。这种方法十分复杂，R 目前也还没有出现与 `knnImputation()` 函数类似的函数，我们在第 6 章将具体地讨论应该如何做。

第 3 章 R 的数据可视化

本章首次接触了 R 中强大的绘图系统，在它的程序包中，专门用于绘画的包就有三四个，而它自身也附带了强大的绘画功能。本章将介绍一些基本统计图形，以及这些图形的作用，并展示如何用函数绘图。这些函数包括基本的绘图函数，也包括一些其他的辅助函数。

3.1 plot() 函数和常用的图形参数

散点图是统计图形中最基本的一种，它反映了两个变量之间的关系。本节将介绍如何用 `plot()` 函数绘制散点图，以及如何使用 `plot()` 自带的参数微调图形，使图形更美观。通过阅读本节，读者将理解 R 中的绘图系统是如何工作的。

3.1.1 设置 `plot()` 函数中的参数

散点图是一种能够反映两个变量之间关系的图形，它在研究变量之间的相关关系时尤其有用。利用 `plot()` 函数能够简洁地画出一个散点图，本节同样选择了一个来自 `datasets` 包的数据集作为演示案例。

```
> head(cars)
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
6     9   10
> plot(cars)
```

数据集 `cars` 中存放了一些有关汽车的数据，使用 `head()` 函数查看 `cars` 的前 6 行数据，由返回结果可知，`cars` 中一共有两个变量，第一个变量存放汽车的速度数据，第二个变量存放汽车的距行驶离数据。在 R 控制台中输入 `plot(cars)` 命令，R 将在控制台右侧的灰色空间中弹出一个如图 3.1 所示的新窗口，该窗口中显示了由 `cars` 中数据制出的散点图。

图 3.1 所示为根据 `plot()` 函数中的默认参数制出的散点图。`cars` 数据集中的第一个变量 `speed` 被默认为散点图的 x 轴，第二个变量 `dist` 被默认为散点图的 y 轴，且 x 轴和 y 轴的名称分别为各自变量的名称。图中的散点都是默认的小圆点，而散点图的坐标

轴也根据 `cars` 数据集中数据的范围做出了相应的设置。显然，这张散点图标出了 `cars` 中所有的数据点，观察这张图片，当 `speed` 增大时，`dist` 也在逐渐增大，`speed` 和 `dist` 之间存在一个较为明显的正相关关系。

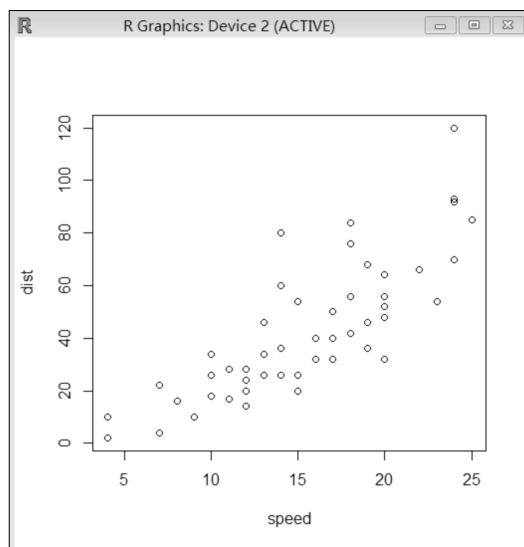
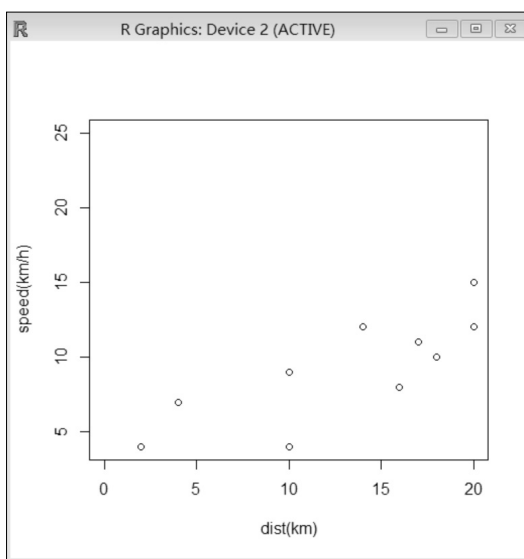


图 3.1 第一个散点图

图 3.2 修改散点图的 x 轴和 y 轴

```
> plot(cars$dist,cars$speed,xlab="dist(km)",ylab="speed(km/h)",xlim=c(0,20))
```

`plot()` 函数中的默认参数大部分都可以修改，上述代码修改了 `plot()` 函数中用于绘制 x 轴、 y 轴的变量，同时也修改了 x 轴和 y 轴的名称，以及 x 轴的范围。在 R 控制台中继续输入上述命令，R 会弹出一个新的绘图窗口，覆盖掉图 3.1 所示的散点图。

上述代码中 `plot()` 函数的前两个参数分别指定了散点图的 x 轴为 `cars` 数据框中的 `dist` 变量, y 轴为 `cars` 数据框中的 `speed` 变量。由于我们并未提前使用 `attach()` 函数绑定 `cars` 数据框, 因此 “`cars$dist`”、“`cars$speed`” 这种书写形式是有必要的, 否则就会报错。

此外, 利用 `plot()` 函数只能绘制二维图片, 这意味着只有当数据框中仅含两个变量时, “`plot(数据框名)`” 这种简洁的命令才能正常工作, 如果数据框中变量数超过了两个, 就需要用 “数据框名 + `$` + 变量名” 的形式指定 x 轴和 y 轴, 否则 R 将搞不清到底该绘制哪两个变量, 同样会报错。

参数 `xlab`、`ylab` 分别修改了 R 中 x 轴和 y 轴的名称, 如果要令散点图的坐标轴名称隐藏起来, 只需设定 `xlab` 或 `ylab` 的值为 "" 即可 (引号中不要填写任何内容)。参数 `xlim` 则修改了 x 轴的坐标范围, 与之对应的还有 `ylim` 参数, 它可以用于修改 y 轴的范围, 这两个参数不但能够缩小图形的坐标范围, 还可以增大图形的坐标范围。

观察图 3.2, 此时该图中坐标轴的名称已经不再是默认值, 而且 x 轴的范围也从 0~120 变为 0~20。但那些处于 20~120 的点并未消失, 它们只是暂时隐藏了起来。另外, `xlab` 参数和 `ylab` 参数的值都需要用双引号引起来, 而 `xlim` 参数则需要给出一个确切的范围。

```
> plot(cars,xlab="speed(km/h)",ylab="dist(km)",main="the data of cars",
+ type="b",pch=4,lwd=2,col="red")
```

上述代码向散点图中增加了主标题, 并修改了散点图中点的默认设置。

如图 3.3 所示, `main` 参数指定了散点图的主标题, 它的命名规则与 `xlab`、`ylab` 相似, 也需要用双引号将标题名引起来, 但当标题名不是字符型, 而是数值型时, 双引号可忽略 (这一规则对 `xlab`、`ylab` 也适用, 但恐怕没有人会想要一个数字作为坐标轴名称)。

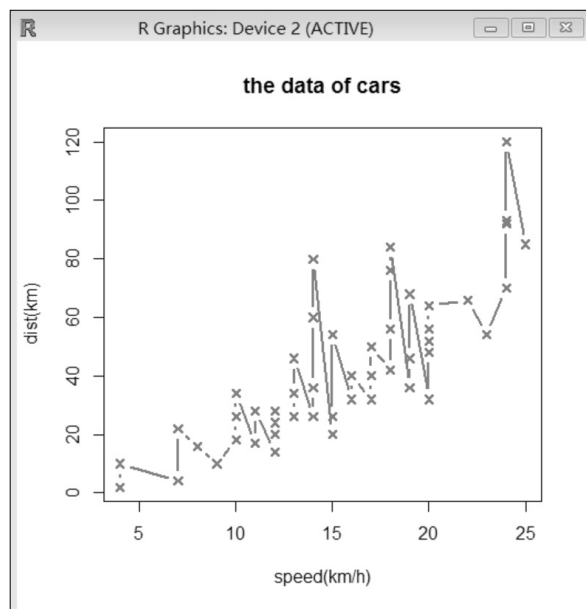


图 3.3 增加标题并修改点

`type` 参数指定了画图的类型,“p”是默认的选项,它画出的是散点图;“l”将画出直线;如本例所示,“b”画出的是由直线连接的点;“c”则会画出贯穿点的直线。其他常用选项还有“o”、“h”、“s”、“S”、“n”等,由于R区分大小写,因此“s”和“S”代表的选项并不一致。

`pch` 参数指定了点的形状,默认值1代表了空心圆圈,如本例所示的4代表小叉,`pch` 参数一直从1规定到25,即 `plot()` 函数提供了25种散点类型以供选择。不过当 `type` 参数指定为“l”时,无论 `pch` 指定成哪个数字,图中都不会有点出现。

`lwd` 参数和 `col` 参数分别指定了点、线的宽度和颜色。`lwd` 参数的默认值为1,参数值越大,线就越粗。但参数值即便设置为小于1的正数,线也不会比1更细。`col` 参数能够指定的颜色有好几百种,`colours()` 命令能查看R中所有可调用的颜色参数。

上述这些参数还不是 `plot()` 函数的全部,在控制台输入“`?plot`”命令可以查看 `plot()` 函数中更多的参数。这些参数在其他的大部分绘图函数中都是适用的,在调用参数时需要注意参数之间要用逗号隔开,参数值的书写格式也不要搞错。

3.1.2 修改散点图的坐标并加入标注

除 `plot()` 函数外,R 同样提供了一些其他的辅助函数用于帮助作图,`axis()` 函数和 `legend()` 函数就是其中较常用的两个函数,与 `plot()` 函数不同,这两个函数并不能自动绘制图片,它们的功能主要在于修饰 `plot()` 函数或其他函数已绘制好的图片。

```
> plot(cars,axes=FALSE)
> axis(1,col="blue",col.axis="red")
> axis(2,lty=2,lwd=0.5)
```

`plot()` 函数并没有提供更细致的修改坐标轴的参数,不过 `axis()` 函数专门提供了一些应用于坐标轴的参数。当 `plot()` 函数中的 `axes` 参数指定为 `FALSE` 时,`plot()` 创建的散点图将没有坐标轴。上述代码首先创建了一个没有坐标轴的散点图,然后在图上逐一添加了两条坐标轴。

R 是一种解释型的语言,在R控制台中输入的每条命令都会立即执行,这意味着在输完第1条代码后,R就会生成一个不带坐标轴的散点图,第2条代码则立即向图中添加一条坐标轴,第3条代码则在已含有一条坐标轴的散点图上再次画出一条坐标轴。最终结果如图3.4所示。

`axis()` 函数中的第一个参数指定了坐标轴的位置,该参数为1时,坐标轴添加在图形的下方;该参数为2时,坐标轴添加在图形的左方;类似地,该参数为3或4时,坐标轴添加在图形的上方或右方。我们在上述代码中简单地写出了一个数字,这也等同于写成“`side=1`”或“`side=2`”的形式,不过只写数字时它必须位于第一个参数的位置,而写成“`side=1`”或“`side=2`”的形式时则不必一定如此。

`axis()` 函数同样可以指定颜色、类型和宽度。在构造散点图下方的坐标轴时,`col` 参数指定坐标轴线的颜色是蓝色,`col.axis` 参数指定坐标轴刻度的颜色是红色。在构造散点图左方的坐标轴时,`lty` 参数指定坐标轴的类型是2,`lwd` 参数指定坐标轴的宽度是0.5。

这些参数与 `plot()` 函数提供的参数都较为类似，`lty` 参数和 `lwd` 参数的可选择范围也与 `plot()` 函数一致。

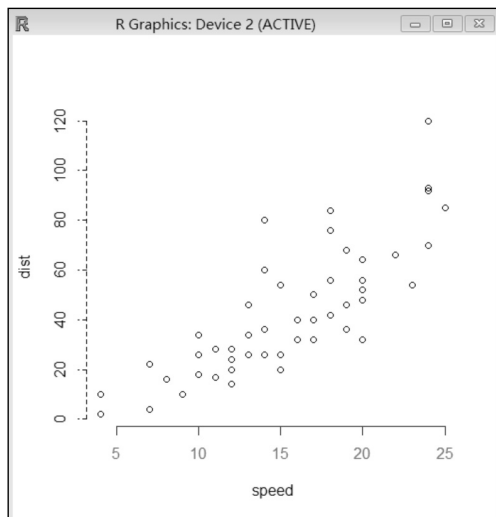


图 3.4 向散点图中添加坐标轴

在控制台中输入 “`?axis`” 命令可查看 `axis()` 函数附带的更多参数，它们所能提供的功能包括调整坐标轴位置、修改坐标轴刻度的细度等。

```
> legend(17,20,"the data of cars",col="orange",pch=1)
> legend(3,105,bquote(sigma[alpha]==0.5),bty="n")
```

`legend()` 函数用于给图形增加图标，上述代码在图 3.4 的基础上，向图片中加入了两个图标。与 `axis()` 类似，这两个图标也是逐一加入散点图中的。代码执行结果如图 3.5 所示。

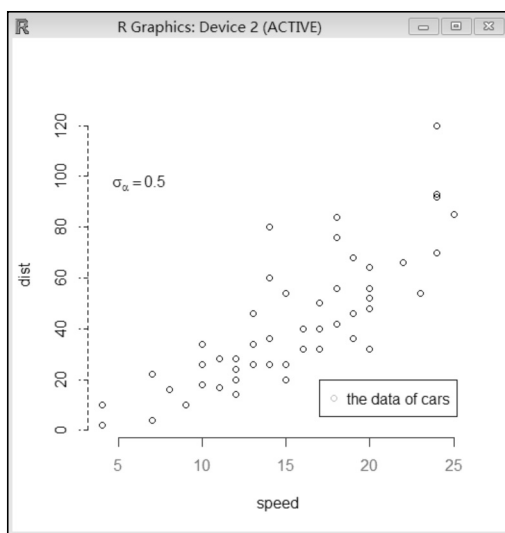


图 3.5 向散点图中加入图标

```
> legend(17,20,"the data of cars",col="orange",pch=1)
> legend(3,105,bquote(sigma[alpha]==0.5),bty="n")
```

`legend()` 函数中前两个参数能够指定图标的位置，与 `side` 参数类似，上述两条代码在指定图标位置时都用了简写的格式。其中参数“17, 20”等同于参数“ $x=17, y=20$ ”；“3,105”等同于参数“ $x=3, y=105$ ”。在设定参数位置时需要多次尝试才能找到最佳的位置，不然图标可能会挡住图中的点，或者超出图形的边界。

上述代码中第1行代码的第3个参数指定图标内容为“the data of cars”，`col` 参数指定图标的颜色是橘色，`pch` 参数则指定图标的类型是1。图中的散点是黑色空心圆圈，而我们设置的图标则是橘色空心圆圈，这其实是不大合理的，在实际工作中，图标也需设置为黑色空心圆圈。

当图形中只有一种点时，设置图标并无多大必要，而当图形中有多种点时，设置图标对点加以区分就是必要的设置。在一个图标中设置多个类型时，只需将第3个参数写为一个形似“`c("x","y",...)`”的字符向量，`col` 参数和 `pch` 参数同样使用 `c()` 函数写成一个与第3个参数等长的字符向量或数值向量，即可得到一系列名称、颜色、符号都不同的图标。

利用 `legend()` 函数同样可以写出专业的数学符号，在上述代码的第2行代码中，`bquote()` 函数将需要输出为数学符号的字符括了起来，显然，“`sigma[alpha]`”被转换成“ σ_α ”后输出到图形中。在 `bquote()` 函数中应用了双等号来输出“`=0.5`”的部分，倘若将双等号写成单等号，R 就会报错。此外，这条命令中的 `bty` 参数指定图标的外框类型为“n”，也就是没有外框。

利用 `legend()` 函数能够很方便地在图形中添加图标，同时 `legend()` 函数对数学字符的广泛支持也增加了 R 中图形的精致程度。在 R 控制台中输入“`?legend`”可以查看 `legend()` 函数提供的更多参数。

3.2 经典的基础图形及用途

继学习过散点图后，本节将介绍4种其他基本统计图形的特点和作用，并展示它们是如何帮助数据分析师进一步探索原始数据的。本节将介绍更多的作图技巧。上一节的内容在本节同样得到了应用。

3.2.1 线图

线图是基本图形的一种。它与散点图相似，经常被用来查看数据的结构。在3.1节中，图3.3展示了用线将点连接后的图形，显然，`cars` 数据集并不适合用来画线图，线图也不能提供更多的有关 `cars` 数据集的有效信息。因此我们选择 `AirPassengers` 数据集来画线图。

```
> head(AirPassengers)
[1] 112 118 132 129 121 135
> plot(AirPassengers,type="l")
```

AirPassengers 数据集中存放了 1949—1961 年飞机运送行李的数据。使用 head() 函数查看 AirPassengers 数据集中的数据，由返回结果可知，这是一列有关行李数量的向量数据，使用 plot() 函数可以简单地画出有关 AirPassengers 的线图。

上述代码中 plot() 函数指定 type 参数为 “l”，这意味着利用 plot() 函数将画出一个线图。由于 AirPassengers 数据集中存放的是一列时间序列类型的数据，因此即便不指定 type 参数，plot() 也同样会为 AirPassengers 自动生成一个线图。

图 3.6 是 R 的返回结果。显然，这是一个典型的时间序列图，图中的曲线呈现一种十分明显的周期性波动，周期长约为一年，每个周期的波动形状都大致相同。同时，随着时间的推移，每个周期的峰值和谷值都在增大，周期内的波动程度也在增加。这说明飞机运送的行李数目随着月份的变化有明显的规律性，同时随着时间推移，人们也越来越多地使用飞机运送行李。

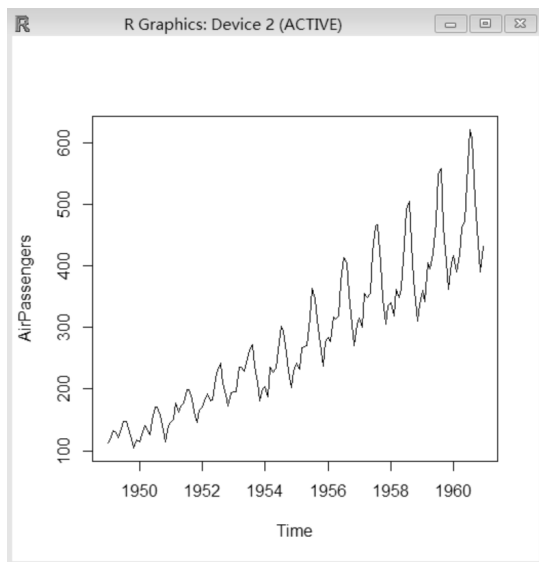


图 3.6 使用 AirPassengers 制作线图

```
> plot(AirPassengers,type="l",xlim=c(1958,1960),ylim=c(300,600),axes=FALSE)
> axis(1,at=seq(1958,1960,1/12))
> abline(v=1958.5)
> abline(v=1959.5)
```

为了进一步观察时间序列图中表现出的规律，上述代码查看了 1958—1960 年内的飞机行李数据，并细化了图中的刻度线，添加了两条辅助线。这些都有助于更好地观察时间序列图。

上述代码中第 1 行代码使用 plot() 函数制作了一张没有坐标轴的线图。其中，xlim 参数和 ylim 参数指定图形的 x 轴为 1958—1960，图形的 y 轴为 300~600。通过放大时间序列图中的这两个周期，我们能够更清楚地看到曲线是怎样变化的。

axis() 函数在图形中添加了一条坐标轴。其中，第一个参数表示这条坐标轴添加在图形的下方，第二个参数 at 表示坐标轴的范围为 1958—1960 年，刻度细度为 0.1。seq()

函数与 `c()` 函数相似，它同样能够生成一个序列，不过不同于仅能生成间隔为 1 的序列的冒号，`seq()` 函数能够指定序列的间隔大小，比如 `seq(1958,1960,1/12)` 所生成的就是一个 1958—1960，间隔为 $\frac{1}{12}$ 的序列，此时图 3.7 中 x 坐标轴的每一个小刻度都代表一个月。

上述代码中的两个 `abline()` 函数分别在 x 坐标为 1958.5 和 1959.5 处添加了一条辅助线，其中 v 参数指定添加竖的辅助线，与之相对的是 h 参数，它可以添加横的辅助线。观察图 3.7，此时图形被两条辅助线分成了三部分，这两条辅助线都基本处于周期的峰值处，两条辅助线夹起的部分是一个完整的周期。

不妨按照一年为一周期的分割方式观察一下图 3.7，显然，每年的一月行李数量有一个较缓的下降，二月行李数量则逐渐上升，经过三、四月的缓和波动后，五、六月是行李数量急速攀升的两个月，曲线在七月到达顶峰，之后便是八、九、十月的急速跌落，曲线在十月达到谷底。这些急速攀升和跌落使得曲线在五月到十月形成了一个尖锐的峰，此后便是十一月、十二月的缓和攀升。`abline()` 函数分别在 1958 年和 1959 年的六月添加了辅助线，基本将这两年一切为二，前半年攀升，后半年下降。

```
> plot(AirPassengers)
> y <- c(1949.5:1960.5)
> a <- function(x){
+   abline(v=x) }
> lapply(y,a)
```

图 3.7 得出的结论是否具有普遍性，这个问题值得注意。上述代码对 1949~1961 年的全体数据均添加了用于观察的辅助线，图 3.8 是最终结果。

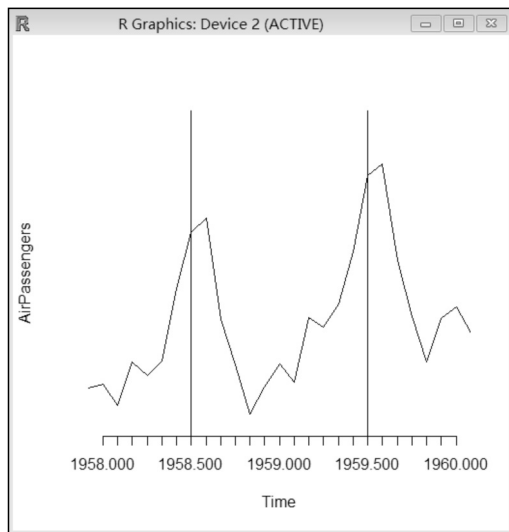


图 3.7 进一步查看时间序列图中的周期

上述代码首先重画了一幅 `AirPassengers` 数据图覆盖掉了图 3.7，然后用三条命令向图中添加了 12 条辅助线。`abline()` 函数仅接受元素作为参数，也就是说，它一次只能画

出一条辅助线。逐一写出 12 个 `abline()` 函数不是不可以，但这样做显然太复杂。上述代码利用 `lapply()` 函数完成了这项任务。

在正式调用 `lapply()` 函数前，上述代码首先创建了一个向量和一个函数。其中，向量 `y` 存放了所有我们想要添加辅助线的 `x` 坐标轴刻度，函数 `a` 则规定对传入函数的参数 `x` 执行代码 `abline(v=x)`，也就是画一条辅助线。`lapply()` 函数将向量 `y` 作为应用对象，对 `y` 中每个元素都应用了函数 `a`，观察图 3.8，此时 12 条辅助线已经完美地全部绘制出来。

`lapply()` 函数的应用对象既可以是向量，也可以是列表，它实现的效果与 `for` 循环语句一致，但它比 `for` 循环要好写一些，其所占用的内存也比 `for` 循环要少。观察图 3.8，此时每一年的六月都添加了一条辅助线，与图 3.7 相似，这些辅助线将每年恰好分成对等的两半，前半年行李数量攀升，后半年行李数量跌落。

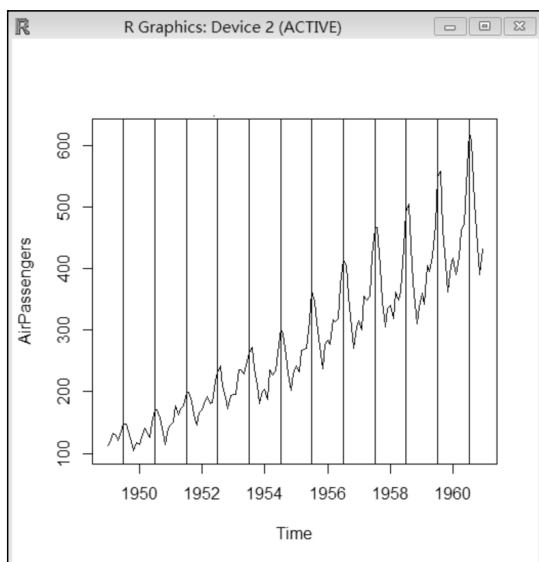


图 3.8 向时间序列图中添加辅助线

```
> lapply(y, function(x) abline(v=x, col="gray"))
```

图 3.8 中的辅助线尽管很有用，但毕竟使图形显得杂乱，上述代码调整了图 3.8 中辅助线的颜色，并演示了 `lapply()` 函数的另一种简洁写法。

上述代码中的 `lapply()` 函数同样具有两个参数，一个参数指定向量 `y` 为应用对象，另一个参数则写出了一个临时函数，其中 `function(x)` 是函数名，`abline(v=x,col="gray")` 是具体的命令块。函数名和命令块之间既没有逗号，也没有花括号。在 `abline()` 函数中，`col` 参数指定辅助线的颜色是灰色。观察图 3.9，图中的 12 条辅助线已经变成一种浅浅的灰色，既提供了辅助信息，同时也不影响对时间序列曲线的整体把握。

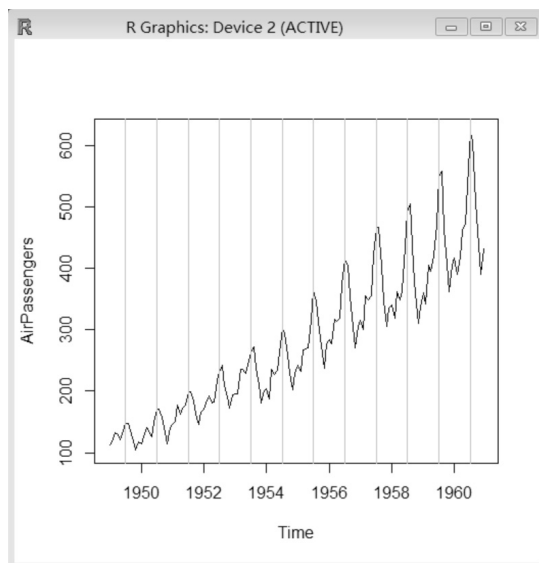


图 3.9 优化 lapply() 函数和时间序列图

3.2.2 直方图

直方图是除散点图和线图外，第三类最常用的统计图形。直方图分为频率直方图和密度直方图，这两种直方图在观察数据集的分布时格外有用。

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4          0.2  setosa
2          4.9         3.0          1.4          0.2  setosa
3          4.7         3.2          1.3          0.2  setosa
4          4.6         3.1          1.5          0.2  setosa
5          5.0         3.6          1.4          0.2  setosa
6          5.4         3.9          1.7          0.4  setosa
> attach(iris)
> hist(Sepal.Width)
```

iris 数据集中存放了 5 列数据，其中前两列数据分别是花萼的长度和宽度，3、4 列数据分别是花瓣的长度和宽度，最后一列数据则存放了花的种类信息。上述代码中第 2 行代码使用 attach() 函数绑定了 iris 数据集，这样在下文中调用 iris 数据集中的变量时会较为方便。

利用 hist() 函数能够直接绘制变量的直方图。图 3.10 所示为利用花萼宽度数据绘制的频率直方图，图中以 0.2 为组距，总共绘出了 12 个长条，每一个长条的高度由落到每一组中数据的多少决定。观察图 3.10，此时 Sepal.Width 的直方图呈现出较为对称的形状，大部分数据都聚集在 3.0 的周围，2.5 以左、3.5 以右的范围内数据较为稀疏。

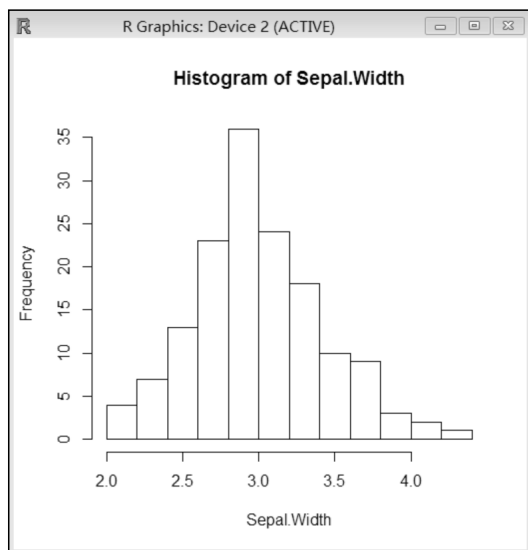


图 3.10 利用花萼宽度数据绘制的频率直方图

除坐标轴名称、标题名称、坐标轴颜色等常用参数外，`hist()` 函数还提供了调整组距的参数和绘制密度直方图的参数。组距会极大地影响直方图的效果，组距过大时，组之间数据的差异将被掩盖，我们不能看出数据究竟聚集在哪一处；组距过小时，直方图又会有太多的峰值，同样影响我们的判断。通常情况下，`hist()` 函数计算出的默认组距较为合适，但当数据是多峰分布，且不同的峰较为接近时，直方图可能会掩盖数据的真实分布情况。

```
> hist(Sepal.Width,breaks=seq(2,4.5,0.00001))
```

`breaks` 参数是 `hist()` 函数中专门用于调整组距的参数，`breaks` 参数有多种设定方法，可以直接写出数字定义直方图的组距，也可以给出一个能够返回组距宽度的函数。最常用的是，如上述代码所示，用一个数字序列来表示组距。

上述代码使用 `seq()` 函数创建了一个数字序列，规定直方图 x 坐标轴的跨度为 2~4.5，组距为 0.000 01。这是一个非常小的组距，观察图 3.11，此时直方图中的小长条已经变成一根根的细线，细线之间的缝隙代表 `Sepal.Width` 中没有这些坐标轴所对应的数据。

这种组距非常小的直方图消去了图中峰值过多时对理解数据的影响，在图 3.11 中，数据仍明显地围绕在 3.0 附近，处于 2.5~3.5 的细线都比较长，此外的细线较短。图 3.11 并未出现多个峰值，也未出现数据明显不对称的情况，因此图 3.10 中给出的默认组距较为合适，不需要更改。

```
> hist(Sepal.Width,freq=FALSE)
> lines(seq(2.0,4.5,0.01),dnorm(seq(2.0,4.5,0.01),mean(Sepal.Width),sd(Sepal.Width)))
> lines(density(Sepal.Width))
```

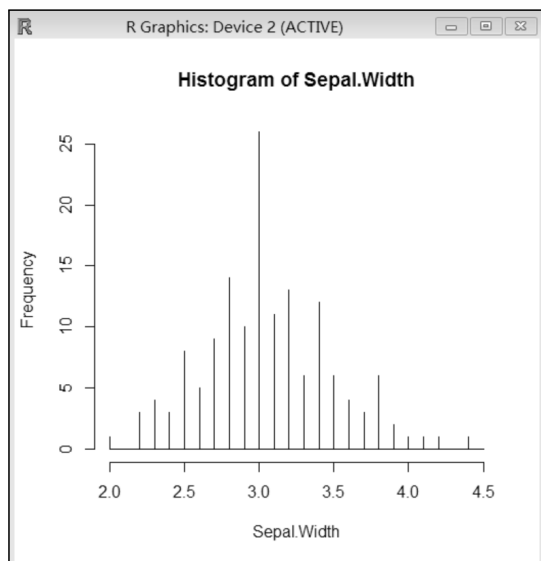


图 3.11 调整花萼数据直方图的组距

密度直方图与频率直方图十分相似，同样能够反映数据的分布情况，当密度直方图与分布曲线相结合时，数据的分布情况能更明白地揭示出来，而密度分布曲线仅能添加在密度直方图上，频率直方图的 y 轴范围与密度分布曲线并不相配。上述代码在 `Sepal.Width` 的密度直方图上添加了自身的分布曲线和正态分布曲线，R 的返回结果如图 3.12 所示。

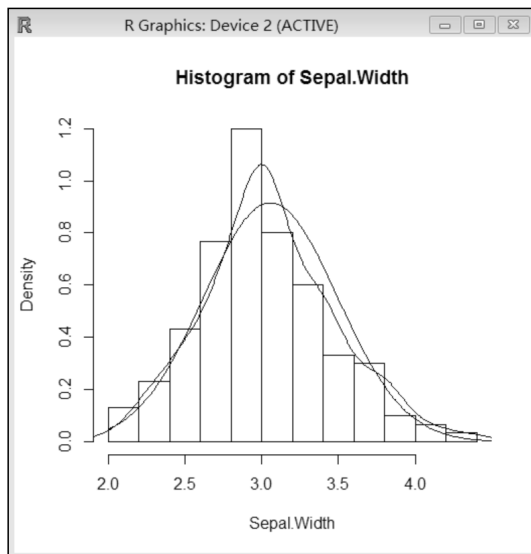


图 3.12 创建密度直方图并添加辅助线

`hist()` 函数中的 `freq` 参数用于指定直方图的类型，当它为 `TRUE` 时，`hist()` 绘制的是频率直方图；当它为 `FALSE` 时，`hist()` 绘制的是密度直方图。图 3.12 所示为一张密度直方图，它与图 3.10 非常相似，但图 3.12 的 y 轴范围已由 0~25 变为 0~1.2，而 y 轴的标题也由 `Frequency` 变为 `Density`。

`lines()` 函数与 `abline()` 函数一致，都用于在图形中添加辅助线。上述代码中的第一个 `lines()` 函数在 `Sepal.Width` 的密度直方图中添加了一条正态分布曲线，它的第一个参数 “`seq(2.0,4.5,0.01)`” 表示 `lines()` 函数将利用这些点绘制曲线，第二个参数 `dnorm()` 则规定了正态分布曲线的细节。其中，第一个参数同样代表 `dnorm()` 将利用这些点绘制正态分布曲线，第二个参数 `mean()` 指明 `Sepal.Width` 的均值将作为正态分布曲线的均值，第三个参数 `sd()` 指明 `Sepal.Width` 的标准差将作为正态分布曲线的标准差。

利用第二个 `lines()` 函数绘制出了 `Sepal.Width` 自身的密度分布曲线，`density()` 函数利用核密度估计法计算了 `Sepal.Width` 的密度分布。在控制台中直接输入 `density(Sepal.Width)` 命令时，R 将返回有关 `Sepal.Width` 密度分布的信息。

图 3.12 中的密度直方图中添加了两条分布曲线，其中光滑对称的、峰点较低的那条曲线是正态分布曲线，而另一条峰点较高的曲线则是 `Sepal.Width` 的密度曲线。比较这两条曲线，显然，与正态分布相比，`Sepal.Width` 中的数据更集中，且 `Sepal.Width` 中的密度均值稍微向左偏，说明 `Sepal.Width` 中较小的一半数据分布得更紧凑，而较大的一半数据分布得更散漫。

3.2.3 箱线图和茎叶图

箱线图和茎叶图都是用于研究数据分布的图形，与直方图相比，这两种图形能更全面地反映数据的分布。2.4 节已经介绍了一些箱线图的特性，但对于箱线图的组合、比较并未提及。本节得介绍更多有关箱线图和茎叶图的内容。

```
> boxplot(iris)
```

利用 `boxplot()` 函数能方便地绘制出箱线图。如图 3.13 所示，上述代码一次性地绘制出了 `iris` 中 5 个变量的箱线图。这种组合图能够非常方便地比较不同变量之间的数据分布，但当变量的量纲相差较大时，这种比较将没有意义，因此对数据进行标准化是有必要的。

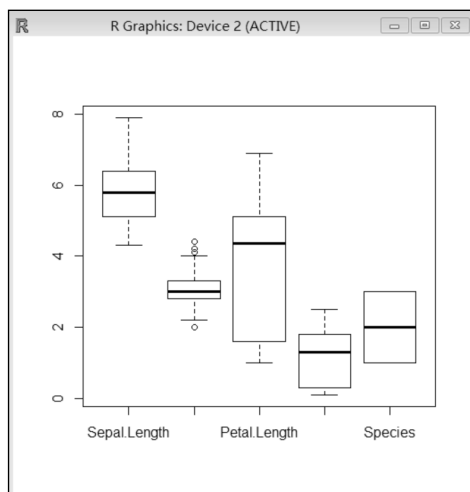


图 3.13 根据 `iris` 绘制箱线图

在这 5 个箱线图中，`Sepal.Width` 的数据最为集中，也只有它有 4 个异常值；`Petal.`

Length 的数据是最为分散的，它的箱子拖得最长，同时，Petal.Length 的中位数十分靠上，也就是说 Petal.Length 中较小的那部分数据十分分散，而较大的那部分数据则较为集中；Sepal.Length 和 Petal.Width 的数据都较为集中，不过 Sepal.Length 的数据是最大的，Petal.Width 的数据是最小的；比较特别的是 Species，它是一个字符型变量，boxplot() 函数为它绘制的箱线图是没有意义的。

```
> boxplot(iris[-5], notch=TRUE, col=c("red", "blue"),
+ names=c("Se.Le", "Se.Wd", "Pe.Le", "Pe.Wd"))
```

图 3.13 中由于变量名称较长，因此 5 个变量的名称并未全部显示出来，上述代码修改了箱线图的名称，并去掉了 Species 箱线图，图 3.14 是它的结果。

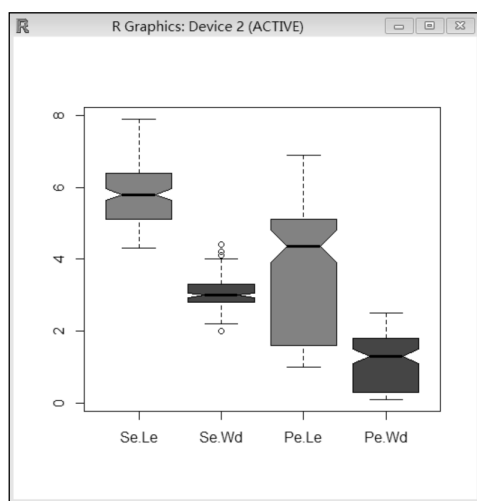


图 3.14 对箱线图进一步优化

上述代码中 boxplot() 函数一共有 4 个参数。其中第一个参数指明 iris 数据集中除第 5 列外的其他变量为绘图对象；第二个参数 notch 指定箱线图是否具有切口，切口标出了距离中位数一倍标准差的位置，当 notch 为 TRUE 时，箱线图中标出切口；col 参数指定了两个颜色，这两个颜色循环着填充了 4 个箱线图；names 参数则为 4 个箱线图命名。

观察图 3.14，此时图 3.13 中第 5 个箱线图已被删去，这 4 个箱线图中新加上的缺口也增加了信息，显然 Pe.Le 和 Pe.Wd 中偏离中位数超过一个标准差的较小数据是这两个变量中最分散的那部分数据。而箱线图的名称经过修改后，图 3.14 也将其完全显示了出来。

```
> stem(Sepal.Length)

The decimal point is 1 digit(s) to the left of the |

42 | 0
44 | 0000
46 | 000000
48 | 000000000000
```

```

50 | 00000000000000000000
52 | 00000
54 | 000000000000000
56 | 0000000000000000
58 | 0000000000
60 | 0000000000000
62 | 00000000000000
64 | 0000000000000
66 | 0000000000
68 | 0000000
70 | 00
72 | 0000
74 | 0
76 | 00000
78 | 0

```

茎叶图是应用于较小数据集上的一类图形，它能够将数据信息完全保留下来。`stem()` 函数用于创建茎叶图，与其他绘图函数不同，茎叶图的返回结果直接返回在 R 控制台中。

对 `Sepal.Length` 变量创建茎叶图，R 首先返回了一句说明：小数点在符号 | 的左一位。观察接下来的茎叶图，图形由两部分组成，一部分在符号 | 左边，它们是一些二位数，称为茎叶图的茎；另一部分在符号 | 的右边，它们是一些 0，称为茎叶图的叶。一个茎对应着多个叶，每一个叶都代表一个具体数据。由于小数点在符号 | 的左一位，因此“42|0”就代表数据 4.20，而“44|0000”中有 4 个叶子，因此它代表 4 个 4.40。

茎叶图中有如此多的 0 显然不方便查看。`stem()` 函数并未设置挪动符号 | 位置的参数，不过通过修改 `scal` 参数同样能够达到相同的效果。

```

> stem(Sepal.Length,scal=0.5)

The decimal point is at the |

4 | 3444
4 | 566667788888999999
5 | 0000000001111111122223444444
5 | 55555556666667777777888888999
6 | 0000001111112222333333334444444
6 | 5555566777777778889999
7 | 0122234
7 | 677779

```

上述代码指定 `stem()` 函数中的 `scal` 参数为 0.5。`scal` 参数的默认值是 1，也就是每个茎对应叶子的范围为 0~9，将 `scal` 参数设置为 0.5 后，每个茎对应叶子的范围将缩减为原来的一半，也就是 0~4，以及 5~9。同样的，`scal` 参数也可以调为比 1 大的数字。

此时 R 的返回结果中小数点恰好在 | 的位置，茎叶图的叶子已经不全是 0，与上一个茎叶图的解读方法类似，这时一片叶子同样代表一个具体数据，比如“4|3444”代表

数据 4.3、4.4、4.4 和 4.4。观察这张茎叶图，显然，Sepal.Length 的数据集中在 5.0~6.9，大于 7 的数据和小于 5 的数据都较少，而 5.5~6.4 的数据又特别多。

3.3 将图形组合起来

在前面我们已经学习了如何使用辅助绘图函数在图形中添加辅助线或辅助点。在关于箱线图的部分也提到了如何在同一幅图中画出多个箱线图。本节讨论的主题是如何将多个不同类型的图形随意组合起来。更确切地说，本章的主题是 `par()` 函数。

`par()` 函数同样是一个辅助绘图函数，与其他辅助绘图函数不同的是，`par()` 函数在绘图之前就提供了一些参数，而不是在图形绘制好以后再对图形进行修饰。

```
> par(mfrow=c(2,3))
> boxplot(cars)
> plot(cars)
> hist(cars$speed)
```

`par()` 函数最重要的作用就是提供能放置多个图形的背景，`mfrow` 参数实现了这个功能。上述代码中第 1 行代码中 `par()` 函数指定 `mfrow` 参数为 `c(2,3)`，也就是创建一个 2 行 3 列的组合图形。在这行代码执行后，R 会弹出一个空白的图形窗口，表示 `par()` 函数已成功执行，接下来只需添加图形即可。

上述代码随后使用 `boxplot()` 函数、`plot()` 函数和 `hist()` 函数分别绘制了箱线图、散点图和直方图。这三个图形依次添加在图形窗口中第 1 行的第 1 列、第 2 列和第 3 列。显然，向 `par()` 函数提供的组合图形窗口中填充图形时，是按照先行后列的顺序。

观察图 3.15 中的图形，它们除小了一些外，其他一切都与单独绘制时没有任何不同。任何在单独绘制图形时起作用的图形参数在此时也是起作用的，如果想对某个图形添加辅助线，只需在绘制好该图形后输入辅助绘图函数即可起作用，不过当下一幅图片绘制后，辅助绘图函数就将对下一幅图片起作用。

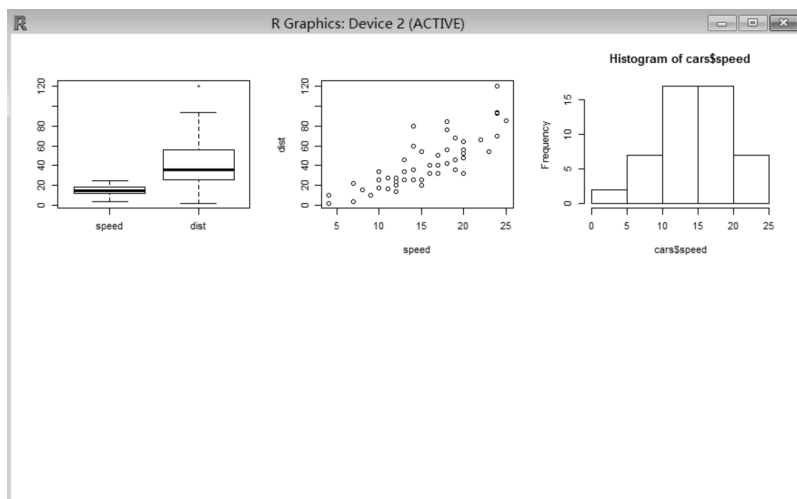
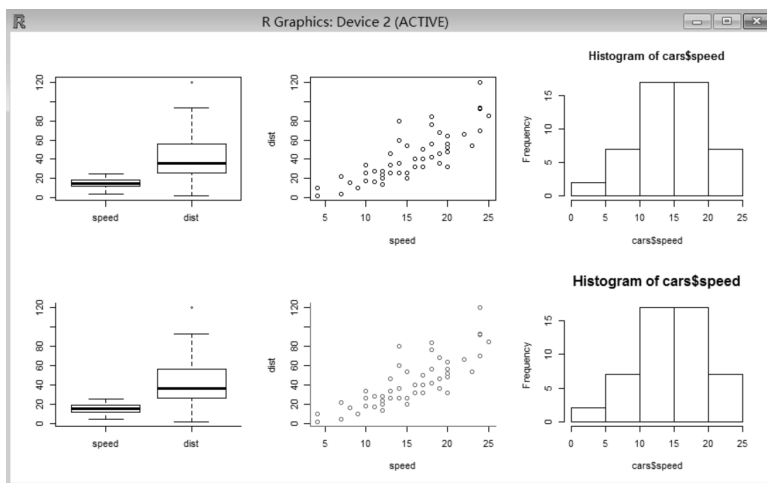


图 3.15 绘制 2 行 3 列的组合图形

```

> par(bty="l")
> boxplot(cars)
> par(col="red")
> plot(cars)
> par(cex.main=1.5)
> hist(cars$speed)

```

图 3.16 在组合图形中使用不同的 `par()` 参数

```

> par(bty="l")
> boxplot(cars)
> par(col="red")
> plot(cars)
> par(cex.main=1.5)
> hist(cars$speed)

```

`par()` 函数同样提供了许多可用的参数。上述代码在接下来的三幅图形中使用了不同的参数。第 1 行代码指定 `bty` 参数为 “l”，`bty` 参数规定了图形框的类型，默认类型 “o” 将图形完全框了起来，示例代码中规定的类型 “l” 则只包住了图形的左边和右边，其他类型还有 “u”、“c”、“n” 等；

第 3 行代码指定 `col` 为红色，观察指定颜色后绘制的散点图，此时图形的坐标轴和散点都变成了红色，同时在第 1 行代码中规定的图形框类型在散点图中也已被应用，`par()` 函数中规定好的参数设置会一直被应用，除非明确修改了参数，或者退出了 R，这个特性有时很方便，有时又很麻烦。

第 5 行代码指定 `cex.main` 为 1.5，即图形的主标题大小为默认设置的 1.5 倍，观察此时绘制的直方图，其标题确实要比之前绘制的大一些。与这个参数类似的还有 `cex.axis`、`cex.lab`、`cex.sub` 等参数。

上述代码在图 3.15 的基础上再次逐个地添加了三幅图形，R 最终将呈现如图 3.16 所示的组合图形。

`par()` 函数还提供了非常多的其他参数，其中大部分参数和其他绘图函数中的参数用

法相似，较为特别的参数有 `mar`、`oma`、`pin`、`fig` 等，它们仅应用于 `par()` 函数，在 R 主页上有更多的相关介绍。

3.4 更多的高水平作图函数

数据分析中常用的统计图形有很多，除散点图、线图、直方图、箱线图和茎叶图外，还有许多其他图形。R 同样提供了作图函数用以绘制这些图形，本节较为概括地介绍了其中同样应用很广泛的三种图形。

```
> qqnorm(AirPassengers)
> qqline(AirPassengers)
```

正态 QQ 图经常被用来判断数据分布与正态分布的接近程度。利用 `qqnorm()` 函数能够很方便地画出数据集的 QQ 图，不过它仅作用于一个变量，而不能作用于多个变量。`qqnorm()` 函数总是和 `qqline()` 函数配合使用，`qqline()` 函数能在 `qqnorm()` 函数绘出的图形中添加一条 QQ 线，它通常总是呈 45 度。

观察图 3.17 所示的正态 QQ 图与正态 QQ 线，图中的空心小圆圈与 `AirPassengers` 中的数据逐一对应，这些小圆圈被 QQ 线紧紧地串在一起，它们与 QQ 线挨得越近，数据分布就越靠近正态分布。由图 3.17 可知，`AirPassengers` 数据与正态分布较为接近。

```
> head(VADeaths)
      Rural Male Rural Female Urban Male Urban Female
50-54    11.7      8.7    15.4      8.4
55-59    18.1     11.7    24.3     13.6
60-64    26.9     20.3    37.0     19.3
65-69    41.0     30.9    54.6     35.1
70-74    66.0     54.3    71.1     50.0
> dotchart(VADeaths)
```

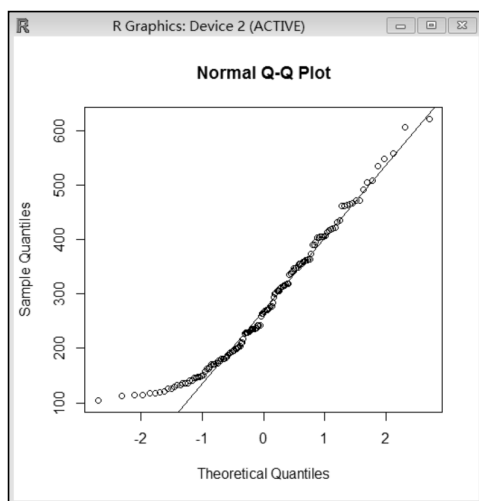


图 3.17 正态 QQ 图与 QQ 线

点图是另一类常用的图形，它的绘制并不复杂，但绘制点图时对数据集的结构有较为苛刻的要求，上述代码使用 `dotchart()` 函数绘制了一张如图 3.18 所示的点图。

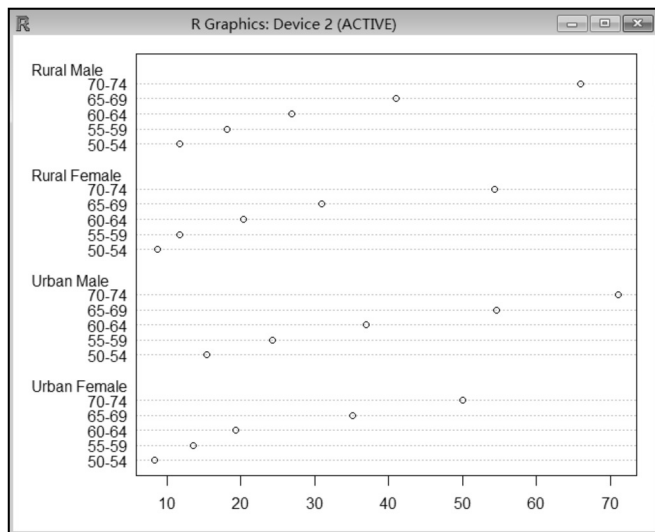


图 3.18 绘制点图

利用 `dotchart()` 函数能够绘制点图，它要求被绘制的数据集必须是向量或者数据框，通常来说，使用数据框绘制点图的效果要好于使用向量。上述代码绘制的点图数据来自 `VADeaths`，使用 `head()` 函数查看 `VADeaths` 中的前 6 行数据，显然，这个数据框不仅标出了列名称，同样还标出了行名称，这些名称将被用于分类，图 3.18 明确地体现了这一点。

观察图 3.18，`VADeaths` 中的 4 列变量将点图分为四大组，每一组又被 5 个行名称区分为 5 个小类。图 3.18 中的每一类都对应了一个点，一条条灰线将这些点和它们对应的小类串了起来。这些点对应的横坐标就是每个小类中包含数据点的个数。点图能很方便地体现包含多个小类的数据集中的信息，但是当数据集不符合这种类型时，点图将不能很好地发挥作用。

```
> par(las=1,mar=c(3,9,1,1),cex.axis=0.53)
> barplot(precip,hORIZ=TRUE,cex.axis=1)
> par(las=1,mar=c(3,9,1,1),cex.axis=0.53)
> barplot(precip,hORIZ=TRUE,cex.axis=1)
```

上述代码首先使用 `par()` 函数规定了一些背景参数，这些参数包括 `las` 为 1，`mar` 为 `c(3,9,1,1)`，以及 `cex.axis` 为 0.53。其中，`las` 参数规定图形纵坐标轴的坐标横向排列，`cex.axis` 参数规定坐标轴的坐标大小为默认大小的 0.53 倍，这两个参数都是为了在纵坐标处挤下全部的名称，`mar` 参数则规定图形的下、左、上、右方分别留下 3、9、1、1 个单位长，这是为了保证横坐标和纵坐标的坐标名称不会超过图形边界。

`barplot()` 函数用于制作条形图，其中，第一个参数指定 `precip` 数据集为数据来源，第二个参数 `horiz` 为 `TRUE` 指定绘制水平条形图，第三个参数则指定坐标轴的坐标大小为默认设置。

观察图 3.19，图中绘制了一个长长的条形图，它的纵坐标名称也非常多，我们不得不把它放到最大，才能看清楚每一个纵坐标名称。条形图的纵坐标被缩小了，但横坐标并没有缩小，这是因为我们在 `par()` 函数和 `barplot()` 函数中分别设置了 `cex.axis` 参数。

条形图是一个大家族，除水平条形图和垂直条形图外，其他常见的条形图还有堆栈条形图和复合条形图。`barplot()` 函数能够绘制出绝大多数的条形图，R 主页同样提供了有关 `barplot()` 函数的更多信息。

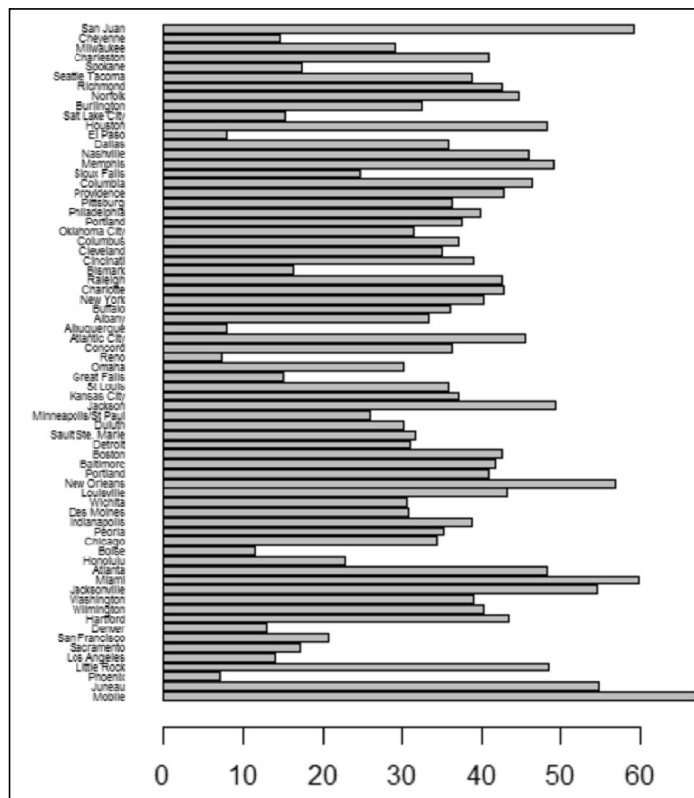


图 3.19 绘制条形图

3.5 更多的常用作图命令

辅助作图函数能够完善绘图函数做出的图形，这种完善一方面弥补了绘图函数的疏漏，另一方面也避免了绘图函数中漏掉参数时不得不再绘一次图的麻烦，在上文中已经提到了两种向图中添加辅助线的绘图函数 `abline()` 函数和 `lines()` 函数，以及向图中添加分布曲线和 QQ 线的 `density()` 函数和 `qqline()` 函数，另外，还有用于修改坐标轴的 `axis()` 函数和用于添加图标的 `legend()` 函数。

除这些函数外，R 还提供了许多其他的辅助作图函数，其中较基础而常见的有添加点和多边形的函数，以及添加标题和文本的函数。

```

> plot(cars)
> points(5,100,col="red",pch=3)
> polygon(c(11,7,13,17),c(5,30,60,10))
> plot(cars)
> points(5,100,col="red",pch=3)
> polygon(c(11,7,13,17),c(5,30,60,10))

```

上述代码首先画出了一张关于 `cars` 数据集的散点图，第 2 行代码中的 `points()` 函数在图中添加了一个点，前两个参数指定这个点的坐标是 (5, 100)，`col` 参数指定该点颜色为红色，`pch` 参数指定该点类型为第三种，观察图 3.20，在坐标 (5, 100) 的地方有一个红色的“+”点，这个点就是 `points()` 函数刚刚添加上去的。

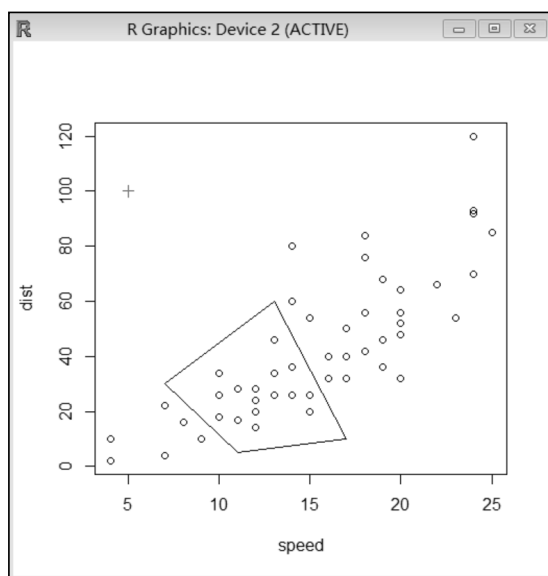


图 3.20 向图中添加点和多边形

`polygon()` 函数用于添加多边形，上述代码中的 `polygon()` 函数中给出了两个长度为 4 的数值向量，它们分别是一列 `x` 轴坐标和一系列 `y` 轴坐标，这两个向量代表了 (11, 5)、(7, 30)、(13, 60)、(17, 10) 4 个点，`polygon()` 函数将这 4 个点连接起来，形成了一个封闭的多边形。

```

> par(mfrow=c(1,2))
> barplot(cars$speed)
> title(main="the main",sub="the sub")
> plot(cars)
> text(10,100,"the text")

```

添加标题的函数是 `title()` 函数，添加文本的函数是 `text()` 函数。上述代码首先用 `par()` 函数创建了一个 1 行 2 列的组合图形。利用 `barplot()` 函数在第一个位置画出了一幅条形图，利用 `title()` 函数在这幅图上添加了标题，其中，`main` 参数添加的是位于图形上方的主标题，`sub` 参数添加的是位于图形下方的子标题。`title()` 函数提供的参数与绘图函

数自带的有关标题的参数并无太大不同，但用 `title()` 函数要较为灵活一些。图 3-21 所示为向图中添加标题和文本。

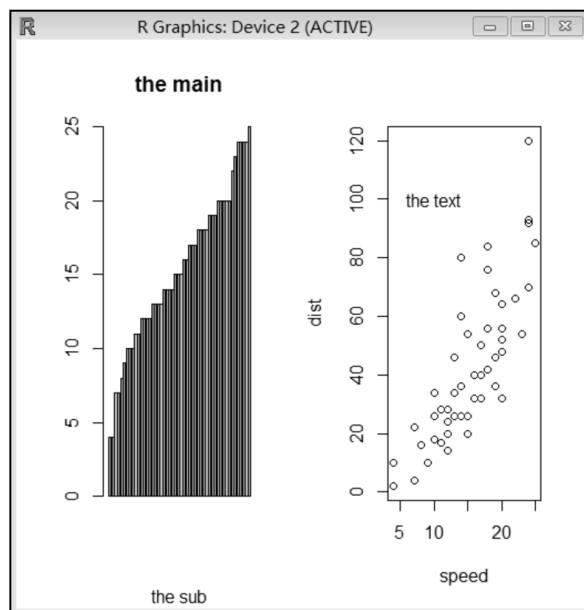


图 3.21 向图中添加标题和文本

利用 `plot()` 函数在第二个位置画出一幅散点图，利用 `text()` 函数在这幅图上添加文本。其中，前两个参数指定文本的坐标为 (10,100)，第三个参数指定文本内容为“the text”。`text()` 函数与 `legend()` 函数的用法十分相似，`text()` 也能够像图形中添加专业的数学符号，但 `text()` 函数通常用于添加一些灵活的文本说明，而 `legend()` 函数则用于添加多种图标。

R 提供了丰富的绘图系统，除本章提到的绘图函数和辅助绘图函数外，R 还能够绘出饼图、雷达图、网格曲面图、轮廓图、调和曲线图等多种图形，图形中也能添加回归直线、其他分布曲线等多种辅助线，第 7 章将系统讲解更多、更高级的图形，在其他章节中，有关绘图的知识也有所补充。

第 4 章 R 中参数的估计和检验

本章的主题为如何使用 R 来估计参数并检验，这是数据分析中较基础的一部分内容，参数的估计和检验也是方差分析、回归分析的基础。通过阅读本章内容，读者将对参数的估计和检验有一个系统的了解，并掌握如何使用 R 进行参数估计与检验。

4.1 使用 R 进行点估计和区间估计

点估计和区间估计是参数估计的两种主要方法。点估计为参数估计出一个确定的值，区间估计则为参数估计出一个区间。本节将比较这两种估计方法的优劣，并用 R 实现参数的点估计和区间估计。

4.1.1 简单的点估计和区间估计

在统计分析中，总体的数据往往是不能全部获得的，很多时候都只能获得一部分样本数据。比如质检中心关心一批新灯泡的寿命均值，但质检中心不能把这批灯泡全部拿来测寿命，只能从中抽取一部分灯泡，这部分灯泡称为样本。质检中心通常用样本数据来估计总体的寿命均值，而这种估计过程就称为参数估计。

参数估计主要分为点估计和区间估计，点估计又分为矩估计、极大似然估计等。矩估计是计算最方便、应用最广泛的一种估计方法，它用样本数据的一阶矩来估计总体的一阶矩，用样本的二阶矩来估计总体的二阶矩，再通过构造方程组来估计总体参数。简单来说，矩估计认为样本的均值近似为总体的均值，样本的方差近似为总体的方差。

```
> BJsales.sam <- sample(BJsales, 50)
> mean(BJsales); mean(BJsales.sam)
[1] 229.978
[1] 228.912
```

BJsales 数据集中有 150 份销售数据，不妨将它看作总体，从中抽取 50 个数据作为样本。sample() 函数能够从总体中随机抽取样本，上述代码中 sample() 函数的第一个参数指定从 BJsales 中抽取数据，第二个参数指定抽取 50 份数据，sample() 函数还提供了 replace 参数，当它为真时，总体中的数据允许被重复抽取。

查看 BJsales 和 BJsales.sam 的均值，这两个均值的差不超过 1.1，此时用样本的均值来估计总体的均值是较为合理的，矩估计法在这里发挥了应有的作用。

```
> 149/150*var(BJsales); var(BJsales.sam)
[1] 458.3010
[1] 438.7488
```

当用样本方差估计总体方差时,矩估计法的误差稍微大了一些,上述代码分别查看了样本和总体的方差。方差的计算公式为 $\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}$,但var()函数将所有被应用对象都看作样本,因此使用var()函数计算方差时,分母是 $n-1$,而非 n 。上述代码中“149/150*var(BJsales)”给出了总体的真实方差。矩估计法认为样本方差与总体方差之间相差一个系数 $\frac{n}{n-1}$,因此“var(BJsales.sam)”计算得到的结果恰好就是根据样本数据估计的总体方差。

观察R返回的结果,总体方差的估计值和实际值的差值约为20,虽然我们知道差值越小,点估计的效果就越好,但显然,很难确定估计值和实际值的差值小到什么地步时才能认为点估计的效果很好。因此,区间估计的引入是有必要的。

样本参数与总体参数之间必然存在一定的联系,但它们又不大可能完全相等。区间估计认为总体参数总是落在样本参数的附近,以样本参数为中心,必然能画出一个区间来覆盖住总体参数。由于样本参数服从的分布是已知的,因此根据置信度和样本参数找出一个置信区间就是可行的。

以均值为例,当总体方差已知时,样本均值服从以总体均值为均值、以总体方差为方差的正态分布,由于样本均值已知,因此可计算出一个置信区间

$$\left[\bar{X} - \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}}, \bar{X} + \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}} \right],$$

其中 \bar{X} 为样本均值, σ 为总体的标准差, n 为样本数据个数,

$Z_{\frac{\alpha}{2}}$ 为标准正态分布的上 α 分位点。当总体方差未知时,样本均值则服从自由度为 $n-1$

的 t 分布,这时置信区间就变为 $\left[\bar{X} - \frac{S}{\sqrt{n}} t_{\frac{\alpha}{2}}(n-1), \bar{X} + \frac{S}{\sqrt{n}} t_{\frac{\alpha}{2}}(n-1) \right]$,其中 S 为样本的标准差, $t_{\frac{\alpha}{2}}(n-1)$ 则为自由度为 $n-1$ 的 t 分布的上 α 分位点。

```
> mean.est <- function(x,sd=-1,alpha){
+ n <- length(x);m <- mean(x)
+ if (sd>=0) {
+ tmp <- sd/sqrt(n)*qnorm(1-alpha/2)
+ }
+ else {
+ tmp <- sd(x)/sqrt(n)*qt(1-alpha/2,n-1)
+ }
+ data.frame(mean=m,sub=m-tmp,sup=m+tmp)
+ }
> mean.est(BJsales.sam,sd=sd(BJsales),alpha=0.05)
      mean      sub      sup
1 228.912 222.9582 234.8658
> mean.est(BJsales.sam,alpha=0.05)
      mean      sub      sup
1 228.912 224.4521 236.9479
```

上述代码根据置信区间的计算公式编写了函数 `mean.est()`，并将 `mean.est()` 函数应用到了样本数据中。`mean.est()` 函数一共有三个形参，第一个参数 `x` 准备传入需要被计算的样本数据；第二个参数 `sd` 准备传入总体数据的标准差，`sd` 已经设置为 `-1`，当总体数据标准差未知时，就不需要再向函数中传入数据；第三个参数 `alpha` 准备传入的是置信度，`alpha` 越大，给出的置信区间就越大，结果就越不可信；`alpha` 越小，给出的置信区间就越小，结果就越可信。

在函数的代码块中 `ifelse` 语句实现了分流，令样本数据按照总体方差已知或未知分别应用不同的计算公式，其中 `qnorm()` 函数和 `qt()` 函数能分别查询正态分布表和 `t` 分布表。函数块的最后一行语句输出了一个由均值、区间左端点、区间右端点构成的数据框。

在设置完均值的置信区间计算函数后，上述代码的最后两行分别调用了该函数，其中第一次调用时给出了总体数据的标准差，第二次调用时并未给出，这两次调用设置的置信度都是 `0.95`。观察 `R` 返回的结果并和 `BJsales` 的均值作比较，显然，这两个置信区间都覆盖了总体均值，当总体方差已知时计算得出的置信区间要小一些，估计效果更好。

估计总体方差的置信区间与估计总体均值类似。当总体均值已知时，总体方差的

置信区间计算公式为 $\left[\frac{n\hat{\sigma}^2}{\chi_{\frac{\alpha}{2}}^2(n)}, \frac{n\hat{\sigma}^2}{\chi_{1-\frac{\alpha}{2}}^2(n)} \right]$ ，其中 $\hat{\sigma}^2$ 表示用总体均值计算得到的样本方差，

$\chi_{\frac{\alpha}{2}}^2(n)$ 和 $\chi_{1-\frac{\alpha}{2}}^2(n)$ 则分别表示自由度为 n 的卡方分布的上 $\frac{\alpha}{2}$ 和 $1-\frac{\alpha}{2}$ 分位数。当总体均值未知

时，总体方差的置信区间计算公式为 $\left[\frac{(n-1)S^2}{\chi_{\frac{\alpha}{2}}^2(n-1)}, \frac{(n-1)S^2}{\chi_{1-\frac{\alpha}{2}}^2(n-1)} \right]$ ，其中 S^2 指样本方差。

```
> var.est <- function(x,m=Inf,alpha){
+ n <- length(x)
+ if (m<Inf) {
+ var <- sum((x-m)^2)/n
+ df <- n
+ }
+ else {
+ var <- var(x)
+ df <- n-1
+ }
+ sub <- df*var/qchisq(alpha/2,df)
+ sup <- df*var/qchisq(1-alpha/2,df)
+ data.frame(var,sub,sup)
+ }
> var.est(BJsales.sam,m=mean(BJsales),alpha=0.05)
      var      sub      sup
1 442.5843 683.9004 309.8453
> var.est(BJsales.sam,alpha=0.05)
      var      sub      sup
1 449.8278 698.5144 313.8822
```


与估计总体均值类似, 上述代码首先创建了一个用于估计总体方差的函数 `var.est()`, 这个函数有三个形参, 其中第一个参数准备传入样本数据; 第二个参数准备传入总体均值, 在这里已经规定了 `m` 为 `Inf`, 即无穷大, 当总体均值未知时, 就不需要再向函数中传入数据; 第三个参数准备传入置信度。函数块同样使用 `ifelse` 语句实现了分支, 其中 `qchisq()` 函数用于查询卡方分布表。

在设置好 `var.est()` 函数后, 上述代码块计算了总体均值已知和总体均值未知两种情况下的置信区间。与 `BJsales` 的实际方差相比, 这两个置信区间都覆盖了真实值, 在总体均值已知时, 置信区间更小, 估计效果更好, 这与计算总体均值时一致, 说明有关总体的信息知道得越多, 对总体参数的估计就越准确。

4.1.2 估计单侧置信区间

与点估计相比, 区间估计不但能估计双侧置信区间, 也能估计单侧置信区间。有时候我们只关心总体参数落在样本参数某一侧的可能性大小, 这时单侧置信区间将发挥作用。与双侧置信区间相比, 单侧置信区间的可能性较大, 计算公式也较多。

对总体均值估计单侧置信区间时, 总体方差已知或总体方差未知时的置信区间

计算公式并不相同。当总体方差已知时, 总体均值的置信区间为 $\left[\bar{X} - \frac{\sigma}{\sqrt{n}} Z_{\alpha}, +\infty\right)$ 或 $\left(-\infty, \bar{X} + \frac{\sigma}{\sqrt{n}} Z_{\alpha}\right]$; 当总体方差未知时, 总体均值的置信区间为 $\left[\bar{X} - \frac{S}{\sqrt{n}} t_{\alpha}(n-1), +\infty\right)$ 或 $\left(-\infty, \bar{X} + \frac{S}{\sqrt{n}} t_{\alpha}(n-1)\right]$ 。

```
> BJsales.sam <- sample(BJsales, 50)
> mean.est <- function(x, sd=-1, side, alpha) {
+   n <- length(x); m <- mean(x)
+   if (sd>=0) {
+     if (side<0) {
+       tmp <- sd/sqrt(n)*qnorm(1-alpha)
+       sub <- -Inf; sup <- m+tmp
+     }
+     else if (side>0) {
+       tmp <- sd/sqrt(n)*qnorm(1-alpha)
+       sub <- m-tmp; sup <- Inf
+     }
+     else {
+       tmp <- sd/sqrt(n)*qnorm(1-alpha/2)
+       sub <- m-tmp; sup <- m+tmp
+     }
+   }
+   else {
+     if (side<0) {
+       tmp <- sd/sqrt(n)*qt(1-alpha, n-1)
```

```

+     sub <- -Inf; sup <- m+tmp
+   }
+   else if (side>0) {
+     tmp <- sd/sqrt(n)*qt(1-alpha,n-1)
+     sub <- m-tmp; sup <- Inf
+   }
+   else {
+     tmp <- sd/sqrt(n)*qt(1-alpha/2,n-1)
+     sub <- m-tmp; sup <- m+tmp
+   }
+ }
+ data.frame(mean=m, sub, sup)
+ }

```

上述代码创建了一个新的 `mean.est()` 函数。与之前的 `mean.est()` 函数相比，新函数增加了一个 `side` 参数，在原有函数的基础上扩展了计算单侧置信区间的功能。当 `side` 参数为负数时，`mean.est()` 函数将计算左侧为负无穷的置信区间；当 `side` 参数为正数时，`mean.est()` 函数将计算右侧为正无穷的置信区间；当 `side` 参数为 0 时，`mean.est()` 函数将计算双侧置信区间。

`mean.est()` 函数的函数块中首先用 `ifelse` 语句分开了总体方差已知与未知的两种情况，在每一种情况下，又嵌套了两组 `ifelse` 语句来判断 `side` 参数与 0 的关系，每一个分支的最末端都由置信区间的计算公式构成。

```

> mean.est(BJsales.sam, side=1, alpha=0.05)
  mean      sub sup
1 230.7 230.4629 Inf
> mean.est(BJsales.sam, side=-1, alpha=0.05)
  mean  sub      sup
1 230.7 -Inf 230.9371
> mean.est(BJsales.sam, side=0, alpha=0.05)
  mean      sub      sup
1 230.7 230.4158 230.9842

```

创建好新的 `mean.est()` 函数后，上述代码查看了 `BJsales.sam` 的两个单侧置信区间及双侧置信区间。此时的双侧置信区间与之前的双侧置信区间已不再一致，这是由于我们在创建新的 `mean.est()` 函数时重新抽取了一组样本数据，因此置信区间也会发生变化。

观察两个单侧置信区间，它们中的实数端点的值与均值的差相同，这是由于计算单侧置信区间宽度的公式是一样的。但这个差值与双侧置信区间端点和均值的差值并不相同，这是由于 t_α 与 $t_{\frac{\alpha}{2}}$ 并不成倍数关系。同样的， Z_α 与 $Z_{\frac{\alpha}{2}}$ 也不成倍数关系。

计算总体方差的单侧置信区间时同样需要区分总体均值是否已知。当总体均值已知时，总体方差的单侧置信区间为 $\left[\frac{n\hat{\sigma}^2}{\chi_\alpha^2(n)}, +\infty \right)$ 或 $\left(-\infty, \frac{n\hat{\sigma}^2}{\chi_{1-\alpha}^2(n)} \right]$ ，当总体均值未知时，总

体方差的单侧置信区间为 $\left[\frac{(n-1)S^2}{\chi_{\alpha}^2(n-1)}, +\infty\right)$ 或 $\left(-\infty, \frac{(n-1)S^2}{\chi_{1-\alpha}^2(n-1)}\right]$ 。

```
> var.est <- function(x,m=Inf,side,alpha){
+   n <- length(x)
+   if (m<Inf) {
+     var <- sum((x-m)^2)/n;df <- n
+   }
+   else {
+     var <- var(x);df <- n-1
+   }
+   if (side<0) {
+     sub <- 0
+     sup <- df*var/qchisq(alpha,df)
+   }
+   else if (side>0) {
+     sub <- df*var/qchisq(1-alpha,df)
+     sup <- Inf
+   }
+   else {
+     sub <- df*var/qchisq(1-alpha/2,df)
+     sup <- df*var/qchisq(alpha/2,df)
+   }
+   data.frame(var,sub,sup)
+ }
```

在构建新的 `var.est()` 函数时，同样增加了一个 `side` 参数用于区分要计算的是单侧置信区间还是双侧置信区间。在函数块中，第一个 `ifelse` 语句用于判断总体均值是否已知，并根据判断结果赋给 `var` 正确的计算公式，第二个分支语句由两小组 `ifelse` 语句构成，用于判断 `side` 参数是否大于 0、小于 0 或等于 0。

当 `side` 小于 0 时，`var.est()` 函数设置的置信区间左端点为 0，而非负无穷，这是因为方差总是大于等于 0，不可能取到比 0 还小的数。在 `var.est()` 函数中并没有嵌套地使用分支语句，这使 `var.est()` 函数的长度要小于 `mean.est()` 的长度。这种写法只有在 `side` 参数不同时计算公式也比较相似的情况下才能采用，否则也缩减不了多少函数的长度。

```
> var.est(BJsales.sam,side=1,alpha=0.05)
      var      sub sup
1 455.9517 336.7815 Inf
> var.est(BJsales.sam,side=-1,alpha=0.05)
      var sub      sup
1 455.9517   0 658.4566
> var.est(BJsales.sam,side=0,alpha=0.05)
      var      sub      sup
1 455.9517 318.1553 708.0238
```

上述代码使用新的 `var.est()` 函数计算了总体均值未知时 `BJsales.sam` 的两个单侧置信

区间和一个双侧置信区间。观察 R 的返回结果，两个单侧置信区间的实数端值与方差值的差并不一致，这是由于它们的计算公式中一个使用的是 $\chi^2_{\alpha}(n-1)$ ，而另一个使用的是 $\chi^2_{1-\alpha}(n-1)$ 。

4.2 与正态总体有关的参数检验

参数检验是参数估计的一个延伸和补充。利用样本参数计算出总体参数的点估计和区间估计后，我们想知道估计值是否能近似看作真实值。参数检验所能做的就是检验某一个参数与另一个参数的关系，仍以 BJsales 数据集为例，参数检验能够给出样本均值与总体均值的相似程度，并且用一个概率值精确度量了这种相似程度，这是点估计和区间估计所做不到的。

```
> BJsales.sam1 <- sample(BJsales,50)
> mean(BJsales)
[1] 229.978
> t.test(BJsales.sam1,mu=229.978,alternative="two.sided")

      One Sample t-test

data:  BJsales.sam1
t = 0.3806, df = 49, p-value = 0.7051
alternative hypothesis: true mean is not equal to 229.978
95 percent confidence interval:
 224.796 237.636
sample estimates:
mean of x
 231.216
```

上述代码仍用 sample() 函数从 BJsales 中抽取了 50 个数据作为样本数据集。从理论上讲，样本均值与总体均值应该十分相似，mean() 函数显示 BJsales 的均值为 229.978，那么 BJsales.sam1 的均值与 229.978 的差异应当非常小，小到可以看作近似相等的程度。

t.test() 函数能够方便地检查样本均值与总体均值之间的差异。上述代码中的 t.test() 函数中提供了三个参数，其中第一个参数指定 BJsales.sam1 为被检验的对象；第二个参数 mu 指定被用来和 BJsales.sam1 的均值作比较的是 229.978，也就是 BJsales 的均值；alternative 指定检验方式为双侧检验，也就是检验 BJsales.sam1 的均值与 229.978 是否相等，如果 alternative 指定为 “less”，t.test() 函数检验的就是 BJsales.sam1 的均值是否大于 229.978，如果 alternative 指定为 “greater”，t.test() 函数检验的就是 BJsales.sam1 的均值是否小于 229.978。

t.test() 函数返回了八九行结果，其中较为重要的是第 3 行给出的 t 值、 df 和 p 值。 t 值是一个由样本均值和总体均值计算得到的统计量，通过查找 t 分布表，能够找到一个 t 值对应的 p 值。 p 值可看作某件事发生的概率，在上述代码中 p 值为 0.705 1，可以认

为 `BJsales.sam1` 的均值与 229.978 相等的概率是 70.51%，这个概率相当高，通常只有 p 值小于 0.05 时，我们才会推翻原假设，认为备择假设成立。 df 给出了数据集的自由度，`BJsales.sam1` 中有 50 个样本，由于 `BJsales.sam1` 的均值已知，因此当 49 个数据已知时，最后一个数据也是已知的，即 `BJsales.sam1` 的自由度为 49。

`t.test()` 函数的第 6 行返回结果同样给出了 `BJsales.sam1` 均值在 95% 置信度下的置信区间。最后一行则给出了 `BJsales.sam1` 的均值。显然，`t.test()` 函数不仅能够检验样本均值与总体均值之间的差异，同样也能检验样本均值与任何一个值之间的差异。通常情况下，如果 `t.test()` 函数检验出一个样本均值与一个总体均值十分接近，那么我们就认为这份样本来自于这个总体。

```
> BJsales.sam2<-sample(BJsales,50)
> t.test(BJsales.sam1,BJsales.sam2,alternative="two.sided",var.equal=TRUE)

      Two Sample t-test

data:  BJsales.sam1 and BJsales.sam2
t = 0.4929, df = 98, p-value = 0.6232
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -6.687522 11.107522
sample estimates:
mean of x mean of y
 231.216   229.006
```

`t.test()` 函数不但能够检验一份数据与一个值之间的差异，同样也能够检验两份数据的均值之间的差异。上述代码首先从 `BJsales` 中再次抽取了一份包含 50 个数据的样本，然后用 `t.test()` 函数检验了这两份样本数据集的均值是否具有差异。

在输入 `t.test()` 函数函数时，第一、二个参数的位置分别放入了两份数据集的名称，第三个参数同样指定检验方法为双侧检验，第四个参数 `var.equal` 则指定两份数据集的方差相同。由于 `BJsales.sam1` 和 `BJsales.sam2` 来自于同一个总体，它们的数据个数也相同，因此这两份数据满足方差相等的条件。`var.equal` 参数的默认值为 `FALSE`，即被检验的两份数据方差不等。

R 再次返回了一长溜结果。其中 p 值为 0.6232，也就是 `BJsales.sam1` 和 `BJsales.sam2` 的均值有 62.32% 的概率相等，这是一个相当高的概率，因此我们接受 `BJsales.sam1` 和 `BJsales.sam2` 的均值相等这一假设。返回结果中也给出了一个置信区间，这个置信区间是两个数据集均值的差的置信区间。在返回结果的最后，`BJsales.sam1` 和 `BJsales.sam2` 的均值都被列了出来。

```
> var.onetest <- function(x,sigma,side){
+   n <- length(x)
+   var <- var(x)
+   df <- n-1
+   chisq2 <- df*var/sigma
+   ld <- length(df)
```

```

+   q <- pchisq(chisq2,df[1])
+   if (side<0) p=q
+   else if (side>0) p=1-q
+   else if (q<0.5) p=2*q
+   else p=2-2*q
+   data.frame(var,df,chisq2,p)
+ }
> var.onetest(BJsales.sam1,sigma=458.3010,side=0)
      var df   chisq2      p
1 465.5021 49 49.47853 0.9080201

```

方差的参数检验同样分为单总体和双总体，R 并未提供检验单总体方差的函数，上述代码创建了用于检验单总体方差的 `var.onetest()` 函数。`var.onetest()` 函数有三个参数，第一个参数 x 用于传入被检验的数据集；第二个参数 `sigma` 用于传入被检验的数值；第三个参数 `side` 用于判断计算单侧 p 值或双侧 p 值。在 `var.onetest()` 函数的函数块中，首先计算了被检验数据集的长度、方差和自由度，以及卡方值 (`chisq2`)。

`pchisq()` 函数计算了 p 值的一个给定值，接下来的 `ifelse` 语句进一步计算了 p 值的最终结果。当 `side` 为负时，计算的是左侧 p 值；当 `side` 为正时，计算的是右侧 p 值；当 `side` 为 0 时，计算双侧 p 值，此时又分 q 小于 0.5 的情况和 q 大于等于 0.5 的情况。

创建好 `var.onetest()` 函数后，上述代码中最后一行代码对 `BJsales.sam1` 数据集应用了该函数，`sigma` 指定为 458.301 0，也就是 `BJsales` 的方差，`side` 函数则指定计算 `BJsales.sam1` 数据集的双侧 p 值。R 返回了 `BJsales.sam1` 数据集的方差、自由度、卡方值和 p 值。其中 p 值为 0.908 020 1，说明 `BJsales.sam1` 的方差有 90.80% 的概率与 `BJsales` 的方差相同，显然，接受 `BJsales.sam1` 的方差与 `BJsales` 相同的假设。

```

> var.test(BJsales.sam1,BJsales.sam2,alternative="less")

      F test to compare two variances

data:  BJsales.sam1 and BJsales.sam2
F = 0.9914, num df = 49, denom df = 49, p-value = 0.488
alternative hypothesis: true ratio of variances is less than 1
95 percent confidence interval:
 0.000000 1.593445
sample estimates:
ratio of variances
 0.9913865

```

R 提供了比较两个数据集的方差的函数。上述代码使用 `var.test()` 函数比较了 `BJsales.sam1` 和 `BJsales.sam2` 的方差，其中第三个参数 `alternative` 指定为 “less”，检验了 `BJsales.sam1` 与 `BJsales.sam2` 的方差比大于 1 的概率。

在 R 的返回结果中 `num df` 给出了 `BJsales.sam1` 的自由度，`denom df` 给出了 `BJsales.sam2` 的自由度， p 值则度量了 `BJsales.sam1` 与 `BJsales.sam2` 的方差比大于 1 的概率，此时 p 值为 0.488，高于 0.05，因此认为 `BJsales.sam1` 和 `BJsales.sam2` 的方差比大于 1。R

同样返回了一个 BJsales.sam1 和 BJsales.sam2 的方差比的置信度为 95% 的置信区间。

var.test() 函数还提供了其他一些有用的参数, 如 ratio 参数用于设定方差比的检验数值, 它的默认值为 1, 此时如果检验通过, 就意味着来自两个数据集的方差相等; conf.level 参数用于设定方差比的置信区间的置信度。其他参数还有 formula、data、subset 参数等。

4.3 列联表与独立性检验

列联表是一类同时具有行名称和列名称的表格, 行和列分别是两组不同的类别, 列联表检验想要检验的是变量的行和列之间是否存在关联, 或者说行数据的增多是否会影响列数据。在第 3 章绘制点图时用到了 VADeaths 数据集, 该数据集就是一个标准的列联表。

```
> VADeaths
      Rural Male Rural Female Urban Male Urban Female
50-54      11.7      8.7      15.4      8.4
55-59      18.1     11.7     24.3     13.6
60-64      26.9     20.3     37.0     19.3
65-69      41.0     30.9     54.6     35.1
70-74      66.0     54.3     71.1     50.0
```

数据集 VADeaths 共有 5 行 4 列, 其中 5 行数据为 5 组年龄, 4 列数据为 4 组性别、居住地不同的居民。在这个列联表中的每一个数据都同时属于两个小组, 比如第 1 行第 1 列的数据 11.7 同时属于 50~54 小组和 Rural Male 小组, 而第 1 行第 2 列的数据 8.7 则同时属于 50~54 和 Rural Female 小组。列联表检验想要探究的就是死亡年龄和居住地、性别是否有关? 住在城市的人的死亡年龄是早于、晚于住在乡村的人, 还是二者一致?

列联表可以看作一个 i 行 j 列的矩阵, 皮尔逊卡方检验是较为常见的列联表检验方法,

卡方检验需要计算 K 值,
$$K = \sum_{i=1}^I \sum_{j=1}^J \frac{\left[n_{ij} - n \left(\frac{n_{i.}}{n} \right) \left(\frac{n_{.j}}{n} \right) \right]^2}{n \left(\frac{n_{i.}}{n} \right) \left(\frac{n_{.j}}{n} \right)},$$
 其中 n_{ij} 为列联表中第 i 行 j 列

的元素, $n_{i.}$ 为列联表中第 i 行的全体元素, $n_{.j}$ 则为列联表中第 j 列的全体元素。将 K 值与 $\chi^2_{\alpha}((I-1)(J-1))$ 作比较, 当 $K > \chi^2_{\alpha}((I-1)(J-1))$ 时, 拒绝原假设, 也就是认为列联表中行元素与列元素有关。在列联表中 $p = P\{\chi^2((I-1)(J-1)) > K\}$, 显然, p 值越大, 列联表中行元素与列元素有关的可能就越小。

```
> chisq.test(VADeaths, correct=FALSE)

Pearson's Chi-squared test

data:  VADeaths
X-squared = 2.9208, df = 12, p-value = 0.9961
```

`chisq.test()` 函数能够很方便地对列联表应用卡方检验。上述代码对 `VADeaths` 应用了 `chisq.test()` 函数，并设定参数 `correct` 为 `FALSE`，也就是不对 `VADeaths` 进行连续校正。观察 `chisq.test()` 函数的返回结果，显然， p 值为 0.996 1，即 `VADeaths` 中居民的居住地、性别和他们的年龄有 99.61% 的概率无关，因此接受原假设，认为这二者无关。

与卡方检验形成互补关系是 Fisher 检验，Fisher 检验专门用于检验某些单元频数小于 5 的列联表，而这正是卡方检验不擅长的地方。此外，Fisher 检验也特别擅长处理 2 行 2 列的列联表。

```
> x<-c(11,9,3,7)
> dim(x)<-c(2,2)
> x
      [,1] [,2]
[1,]   11   3
[2,]    9   7
```

`VADeaths` 并没有小于 5 的单元频数，上述代码构建了一个 2 行 2 列的小矩阵 `x`，`fisher.test()` 函数能够将矩阵看作列联表进行处理。

```
> fisher.test(x)

      Fisher's Exact Test for Count Data

data:  x
p-value = 0.2602
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4554321 21.4445394
sample estimates:
odds ratio
 2.752681
```

上述代码将矩阵 `x` 传入 `fisher.test()` 函数，对 `x` 进行了 Fisher 检验。R 的返回结果显示 Fisher 检验的 p 值为 0.260 2，`x` 的行数据和列数据有 26.02% 的概率无关，同时行变量和列变量比值的置信区间大约为 0.45~21.44，其中包含了 1，因此 `x` 的行数据和列数据是无关的。

4.4 几种检验数据分布的函数

在前面讨论了有关点估计、区间估计和参数检验的内容，但这些估计方法和检验方法中，除去卡方检验和 Fisher 检验外，其他检验方法都只对正态分布起作用，当数据不满足正态分布时，上述参数估计和检验得到的结果将不具有意义，此时就需要对数据使用其他的检验方法。

正态分布是自然界中最常见的一种分布，除正态分布外，其他常见的分布还有均匀分布、指数分布、泊松分布等。显然，找出数据究竟服从什么分布是非常重要的，R 提

供了好几种函数用于检验数据的分布。

Shapiro-Wilk 检验专门用于检验数据是否服从正态分布。它能够为数据集计算出一个 W 统计量, 将该 W 统计量与临界值相比较, 即可得到度量数据是否服从正态分布的 p 值。R 中有关 Shapiro-Wilk 检验的函数是 `shapiro.test()` 函数。

```
> attach(iris)
> shapiro.test(Sepal.Width)

      Shapiro-Wilk normality test

data:  Sepal.Width
W = 0.9849, p-value = 0.1012
```

`shapiro.test()` 函数只有一个用于指定被检验对象的参数, 这个参数只能传入数值型向量, 数据框和列表都是不合法的。上述代码首先绑定了 `iris` 数据框, 然后对 `iris` 中的 `Sepal.Width` 向量应用了 `shapiro.test()` 函数, 尽管 `iris$Sepal.Width` 与 `iris[1]` 都指向 `iris` 中的第一列数据, 但倘若写作 `shapiro.test(iris[1])` 就是不合法的, 这是由于 `iris$Sepal.Width` 是一个向量, 而 `iris[1]` 仍是一个数据框。

R 返回了有关 Shapiro-Wilk 检验的统计值, 其中 p 值为 0.101 2, 它大于 0.05, 因此不能拒绝原假设, 即不能认为 `Sepal.Width` 不服从正态分布, 但是 p 值也没有特别大, 因此我们也不会接受原假设, 认为 `Sepal.Width` 服从正态分布。

```
> x<-c(1:5)
> y<-seq(2.0,4.5,0.5)
> y
[1] 2.0 2.5 3.0 3.5 4.0 4.5
> for (i in 1:5)
+ x[i]<-sum(Sepal.Width<y[i+1])-sum(Sepal.Width<y[i])
> x
[1] 11 46 68 21 4
```

`chisq.test()` 函数不但能够检验均值, 同时也提供了检验数据分布的参数, 但与 `shapiro.test()` 函数不同, `chisq.test()` 函数接受的不是全体数据, 而是全体数据在特定区间内的频数。`Sepal.Width` 的数据分布在 2.0~4.5, 我们将它均匀地分为 5 个区间, 上述代码统计了这 5 个区间的频数。

上述代码首先创建了一个长度为 5 的向量 `x`, 并创建了一个从 2.0 到 4.5 的间隔为 0.5 的序列 `y`, 显然, `y` 中存放的是 5 个区间共有的 6 个端点。`sum()` 函数并不能直接计算出某个区间内的频数, 因此接下来的 `for` 循环实现了统计区间内频数的作用。`for` 循环的循环名表明 `i` 从 1 到 5 循环 5 次, 每一次都执行 `for` 循环的循环块, 每一次循环都将区间 `(y[i],y[i+1])` 内 `Sepal.Width` 的频数赋给 `x[i]`。由于循环块只有一条语句, 因此并未用花括号括起, 若循环块多于一句, 就需要用花括号将循环块括起来。查看 `x` 向量, 此时 `x` 中已经存放了 5 个元素, 分别对应着区间 (2.0, 2.5]、(2.5, 3.0]、(3.0, 3.5]、(3.5, 4.0]、(4.0, 4.5] 内 `Sepal.Width` 的频数。

```
> chisq.test(x,p=rep(1/5,5))

Chi-squared test for given probabilities

data:  x
X-squared = 93.9333, df = 4, p-value < 2.2e-16
```

上述代码中的 `chisq.test()` 函数使用了一个新的参数 p , p 用于指定被检验数据集的分布函数, p 给出的概率分布的总和默认为 1。上述代码指定的分布函数为均匀分布, `rep()` 函数用于重复值, `rep(1/5,5)` 给出的序列就是 0.2, 0.2, 0.2, 0.2, 0.2。这 5 个值与 x 中的 5 个频数相对应, 意味着 `chisq.test()` 检验的是 `Sepal.Width` 中的数据在 5 个小区间内出现的概率是否均为 0.2。

由 R 的返回结果可知, p 值小于 2.2^{-16} , 这个值非常小, 因此认为 x 与 `rep(1/5,5)` 并不相似, 也就是 `Sepal.Width` 不服从均匀分布。

`chisq.test()` 函数要求输入数据集的区间频数形式, 分布函数也需要计算概率并指定。`chisq.test()` 函数能够检验数据集是否服从其他常见的分布, 但需要用户熟悉被检验分布的分布形式, 并手动输入。与 `chisq.test()` 函数相比, `ks.test()` 函数要简洁一些。

```
> ks.test(Sepal.Width,"ppois",1/50)

One-sample Kolmogorov-Smirnov test

data:  Sepal.Width
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

当检验单个数据集是否服从某种分布时, `ks.test()` 函数必须指定的参数有被检验的数据集名称、被检验的分布名称及被检验分布的参数。上述代码中 `ks.test()` 函数的参数 "ppois" 指定检验泊松分布, 参数 1/50 则指定检验的泊松分布是参数为 0.02 的泊松分布。

`ks.test()` 函数能够检验绝大部分的分布。在检验其他分布时, 有多少个参数就需要指定多少个参数, 比如检验指数分布和均匀分布时, 需要指定一个参数, 而检验 F 分布时, 就需要指定两个参数, 写为 `ks.test(x,"pf",y,z)` 的格式, 其中, x 处填写被检验数据集的名称, y 和 z 处分别填写 F 分布的两个参数。

观察 R 的返回结果, p 值同样是一个非常小的值, 因此接受备择假设, 不认为 `Sepal.Width` 服从参数为 0.02 的泊松分布。

```
> ks.test(Sepal.Width,Petal.Width)

Two-sample Kolmogorov-Smirnov test

data:  Sepal.Width and Petal.Width
D = 0.9067, p-value < 2.2e-16
alternative hypothesis: two-sided
```

`ks.test()` 函数也能检验两个数据集是否具有相同分布。上述代码检验了 `Sepal.Width` 是否和 `Petal.Width` 的分布相同，由于此时并未指定分布函数，因此也不需指定分布函数的参数。观察 R 的返回结果， p 值要远远小于 0.01，因此这两个数据集的分布是不相同的。

R 的返回结果还指出了 `alternative hypothesis`（备择假设）是双边的，`ks.test()` 函数提供了 `alternative` 参数用于指定检验双侧分布或单侧分布，其用法与 `t.test()` 等函数一致。

4.5 对非正态总体的区间估计和检验

4.4 节介绍了几种检验数据分布的函数，我们最关心的是数据是否服从正态分布，对不服从正态分布的数据，大部分参数检验的假设不能满足，此时就需要进行不对数据分布做出要求的非参数检验。

4.5.1 非正态总体的区间估计

点估计对非正态总体同样起作用，但 4.1 节提到的区间估计并不能直接用于非正态总体。根据中心极限定理，此时样本数据与样本均值的差的和与标准差的商服从正态分布。因此，非正态总体的均值置信区间为 $\left[\bar{X} - \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}}, \bar{X} + \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}} \right]$ ，当总体方差已知时， σ 就是总体的标准差；当总体方差未知时， σ 就是样本的标准差。

```
> mean.est <- function(x,sd=-1,alpha){
+   n <- length(x)
+   m <- mean(x)
+   if (sigma<0)
+     tmp <- sd(x)/sqrt(n)*qnorm(1-alpha/2)
+   else
+     tmp <- sd/sqrt(n)*qnorm(1-alpha/2)
+   data.frame(mean=m,sub=m-tmp,sup=m+tmp)
+ }
```

上述代码根据非正态总体的均值置信空间创建了函数 `mean.est()`，这个函数有三个参数，其中第一个参数 `x` 准备传入被估计的样本数据；第二个参数 `sd` 准备传入总体标准差，它默认设置为 -1，因此，当总体标准差未知时，无须设置这一参数；第三个参数 `alpha` 准备传入置信度，与之前的例子相同，`alpha` 设置得越小，置信区间就越小，结果就越可信，`alpha` 越大，置信区间就越大，结果就越不可信。

在函数块内，`mean.est()` 首先求得了样本数据集的长度 n 和均值 m ，然后用一组 `ifelse` 语句对总体方差已知或未知的两种情况进行了分流，并对每种情况分别计算了置信区间的宽度，最后返回了一个由样本均值、置信区间下端点、置信区间上端点构成的数据框。

```
> attach(iris)
> sam <- sample(Sepal.Length,100)
```

```
> mean.est(sam, sd=sd(Sepal.Length), alpha=0.05)
      mean      sub      sup
1 5.847 5.684702 6.009298
> mean.est(sam, alpha=0.05)
      mean      sub      sup
1 5.847 5.671045 6.022955
```

创建好 `mean.est()` 函数后, 上述代码对 `iris` 中的 `Sepal.Length` 向量应用了该函数, 为了方便调用数据, `attach()` 函数首先绑定了 `iris` 数据集。`sample()` 函数从 `Sepal.Length` 中取了 100 个数据作为样本数据 `sam`。最后两行代码对 `sam` 数据集分别在总体方差已知和总体方差未知这两种情况下应用了 `mean.est()` 函数。当总体方差已知时, 总体均值的置信区间为 `[5.684 702, 6.009 298]`; 当总体方差未知时, 总体均值的置信区间为 `[5.671 045, 6.022 955]`。显然, 总体方差有利于推断总体均值。

对非正态分布的数据集估计置信区间的步骤与正态分布数据集相似, 它同样也能细分为单侧置信区间估计, 以及对其他参数的置信区间估计, 只是计算公式并不相同。此外, 非正态分布数据集的置信区间估计公式利用中心极限定理, 因此, 使用该公式时还要求样本数据要足够多, 一方面是指样本数据的个数足够多, 另一方面是指样本数据在总体数据中所占的份额要大。

针对非正态分布构造的置信区间估计公式的效率不如针对正态分布构造的置信区间估计公式高。中心极限定理对任何分布都适用, 因此 $\left[\bar{X} - \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}}, \bar{X} + \frac{\sigma}{\sqrt{n}} Z_{\frac{\alpha}{2}}\right]$ 也能够用于估计正态分布的总体均值, 实际上当总体方差已知时, 这两种方法的结果一模一样; 但当总体方差未知时, 该置信区间就不如 $\left[\bar{X} - \frac{S}{\sqrt{n}} t_{\frac{\alpha}{2}}(n-1), \bar{X} + \frac{S}{\sqrt{n}} t_{\frac{\alpha}{2}}(n-1)\right]$ 估计的效果好。

4.5.2 非参数检验中的符号检验

符号检验是最简单的一种非参数检验, 它能检验一个样本是否来自某个总体, 或检验两个总体是否存在差异。符号检验使用正、负号标注数据, 并根据正、负号的多少进行判断。

比如, 当用符号检验检验样本数据 “35” 是否来自 “22, 67, 52” 这一总体时, 符号检验会比较样本数据与总体数据的大小, 得到 “+, -, -” 这一组符号, 并根据这组符号中正、负号的多少来判断 “35” 是否来自该总体。这种思想在检验两个总体之间是否存在差异时也同样适用。

```
> PL.m <- mean(Petal.Length)
> PL.m
[1] 3.758
> binom.test(sum(Sepal.Length<PL.m), length(Sepal.Length), alternative="less")

Exact binomial test
```

```
data: sum(Sepal.Length < PL.m) and length(Sepal.Length)
number of successes = 0, number of trials = 150, p-value < 2.2e-16
alternative hypothesis: true probability of success is less than 0.5
95 percent confidence interval:
 0.00000000 0.01977344
sample estimates:
probability of success
0
```

R 中有关符号检验的函数是 `binom.test()` 函数，上述代码检验了 `Sepal.Length` 向量是否普遍小于 `Petal.Length` 的均值。在上述代码中，`PL.m` 中存放了 `Petal.Length` 的均值，`binom.test()` 函数中一共指定了三个参数。第一个参数使用 `sum()` 函数求出了 `Sepal.Length` 中所有小于 `PL.m` 的数据的个数，也就是在这个符号检验中所有符号为“+”的个数；第二个参数给出了 `Sepal.Length` 的长度，也就是这个符号检验中所有符号的个数；最后一个参数则指定检验 `Sepal.Length` 向量是否普遍小于 `Petal.Length` 的均值。

查看 R 的返回结果， p 值是一个相当小的数，因此拒绝原假设，认为 `Sepal.Length` 向量不是普遍小于 `Petal.Length` 的均值的。R 同样返回了一个置信度为 95% 的置信区间，0.5 并不被区间包含，也说明 `Sepal.Length` 向量并不普遍小于 `Petal.Length` 的均值，显然，这吻合了自然界中花瓣往往要比花萼长这一事实。

```
> binom.test(sum(Sepal.Length<Petal.Length),length(Sepal.Length))

Exact binomial test

data: sum(Sepal.Length < Petal.Length) and length(Sepal.Length)
number of successes = 0, number of trials = 150, p-value < 2.2e-16
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.00000000 0.0242926
sample estimates:
probability of success
0
```

符号检验同样能检验两个总体是否存在差异，此时符号检验要求两个总体之间具有一一对应的关系，即 A 总体和 B 总体要有同样的长度，且分布来自两个总体第 i 个位置的两个数据之间具有某种联系。不妨设 `Sepal.Length` 按照顺序存储了编号从 1 到 150 的 150 朵花的花瓣长度，`Petal.Length` 同样按照顺序存储了这 150 朵花的花萼长度，则此时向量 `Sepal.Length` 和 `Petal.Length` 的同一位置上就存放了来自同一株花朵的花萼长度和花瓣长度，这两个向量就具有了一一对应的关系，能够进行符号检验。

上述代码为 `binom.test()` 函数同样指定了两个参数，第一个参数统计了 `Sepal.Length` 中所有小于对应的 `Petal.Length` 值的数据的个数，也就是符号检验中“+”号的个数；第二个参数记录了符号检验中所有符号的总数；此时并未指定 `alternative` 参数，则默认检验 `Sepal.Length` 向量是否与 `Petal.Length` 向量相同。

由 R 的返回结果可知, p 值是一个相当小的数, 因此 Sepal.Length 向量与 Petal.Length 向量是不同的, 也就是一朵花的花瓣和花萼并不等长。R 返回的置信度为 95% 的置信区间同样拒绝原假设。

4.5.3 非参数检验中的秩检验

秩检验是一种用来度量两样本相关性的检验, 它同样要求被检验的两个样本具有一一对应的关系, 并且可以细分为斯皮尔曼秩检验或肯达尔秩检验。在构建统计量时, 秩检验同样不需要有关数据分布的参数, 因此, 它也是非参数检验的一种, 不对数据集的分布做出要求, 适用于所有的非正态分布。

斯皮尔曼秩检验使用斯皮尔曼秩相关系数作为检验标准。斯皮尔曼秩相关系数的计算公式为 $r_s = \frac{\sum_{i=1}^n (R_i - \bar{R})(S_i - \bar{S})}{\sqrt{\sum_{i=1}^n (R_i - \bar{R})^2} \sqrt{\sum_{i=1}^n (S_i - \bar{S})^2}}$, 其中 R_i 、 S_i 代表来自两个样本的数据点, \bar{R} 、 \bar{S}

则代表两个样本的均值。 r_s 显然是一个处于 -1 到 1 之间的系数, 该值为正数时, 两个样本具有正相关关系; 该值为负数时, 两个样本具有负相关关系, 该值的绝对值越接近 0, 就表示两个样本的相关关系越弱。

```
> cor.test(Sepal.Length, Petal.Length, method="spearman")

Spearman's rank correlation rho

data: Sepal.Length and Petal.Length
S = 66429.35, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.8818981
```

R 提供了 cor.test() 函数来进行秩检验, 上述代码的 cor.test() 函数共有 4 个参数, 前两个参数指定被检验的两个数据集为 Sepal.Length 和 Petal.Length, 第三个参数指定检验方法为斯皮尔曼秩检验。由于并未指定 alternative 参数, 此时默认的是双侧检验, 即检验 Sepal.Length 和 Petal.Length 是否具有相关关系。

在 R 的返回结果中 p 值十分小, 因此拒绝原假设, 认为这二者不具有相关关系。同时 R 还返回了 r_s 值, 该值为 0.881 898 1, 意味着 Sepal.Length 和 Petal.Length 有很强的正相关关系, 当然, 这个结论只有在 p 值大于 0.05 时才有意义。

肯达尔秩相关系数是一个与斯皮尔曼秩相关系数非常相似的系数。它同样用于度量两个具有对应关系的样本 A、B 的相关程度, 称 $(A[i], B[i])$ 为一个观测值, 两个观测值称为一个数对。肯达尔秩相关系数中规定当 $A[i]$ 大于 $A[j]$ 且 $B[i]$ 大于 $B[j]$ 时, $(A[i], B[i])$ 和 $(A[j],$

$B[j]$) 就称为一个协同数对。肯达尔秩相关系数的计算公式为 $\tau = \frac{2 \times (N_c - N_d)}{n(n-1)}$, 其中 N_c 表示样本数据中所有协同数对的数目, N_d 表示样本数据中所有不协同数对的数目。显然 N_c 与 N_d 的和为 $\frac{n(n-1)}{2}$ 。

```
> cor.test(Sepal.Length, Petal.Length, method="kendall")

Kendall's rank correlation tau

data: Sepal.Length and Petal.Length
z = 12.6465, p-value < 2.2e-16
alternative hypothesis: true tau is not equal to 0
sample estimates:
      tau 
0.7185159
```

肯达尔秩相关系数也使用 `cor.test()` 函数加以检验, 上述代码再次检验了 `Sepal.Length` 与 `Petal.Length` 的相关性, `method` 参数指定使用肯达尔秩相关系数进行检验。在 R 的返回结果中 p 值仍是一个极小的数值, 因此不能认为 `Sepal.Length` 和 `Petal.Length` 具有相关关系。肯达尔秩相关系数同样是一个处于 -1 到 1 之间的数, `Sepal.Length` 和 `Petal.Length` 的肯达尔秩相关系数为 $0.718\ 515\ 9$, 表明这两个向量具有很强的正相关关系, 不过, 这个值只有在 p 值大于 0.05 时才可信, 否则是不可信的。

第5章 R 中的方差分析

方差分析是统计试验设计中相当重要的内容，它关心的是一个数值型变量与多个分类型变量是否存在关系。方差分析在农学、医药学、工程学等需要大量实验设计的社会学科中应用十分广泛。本章围绕方差分析展开，将由简到难地介绍4种常见的方差分析模型，并介绍与之类似的多样本非参数检验。

5.1 方差分析模型的建立

第4章已经讨论过了如何利用t检验观察两个正态总体是否具有相同分布。从功能上讲，方差分析可以视为t检验的升级版，它观察多个正态总体是否具有相同分布。但方差分析的假设条件要比t检验严苛一些，方差分析的应用也远远广于t检验。

在建立方差分析模型时，将每一个数据称为一个观测值，将数值变量称为观测变量，将分类变量称为控制变量，比如，在种植实验中在同一种土地上分别种植同样多的5种农作物，这5种农作物的产量就是观测变量，5种农作物的种类就是控制变量，显然，我们关心的是控制变量的变化是否会影响观测变量，以及控制变量对观测变量的影响的强弱和好坏。

方差分析有三条假设，首先，方差分析模型是线性可加模型，模型中的分量之间应该具有线性关系，也就是总体的自变量和因变量能写成 $x_{ij} = \mu + \sigma_i + \varepsilon_{ij}$ 的形式，其中 x_{ij} 代表第 i 个总体中的第 j 个观测值， μ 代表总和的均值， σ_i 代表第 i 个总体对观测变量的效应， ε_{ij} 代表该观测值的误差；其次，方差分析模型中涉及的多个总体都应该服从正态分布；最后，不同总体应该具有相同的方差。

大部分实验数据都满足这三条假设，对于不满足正态分布假设的数据来说，非参数检验中的秩检验能够达到同样的效果；对于不满足方差相同假设的数据来说，标准化数据、转换数据或清洗数据能够改善数据质量。

在方差分析中，将控制变量称为因素，控制变量的具体种类为不同的水平。当观测变量和控制变量都只有一个时，构建的方差分析模型就是最简单的单因素方差分析模型；当控制变量的个数为两个时，方差分析模型又称为双因素方差分析模型，比如同时研究父母的学历和生育年龄是否会影响孩子的智力；当观测变量多于一个时，方差分析模型又称为多元方差分析模型，比如研究父母的生育年龄是否会影响孩子的智力和学历。

双因素方差分析涉及两个控制变量，这两个控制变量是否具有相关关系也会影响分析结果。比如父母的生育年龄也许并不会影响孩子智力，但由于父母的学历会影响生育年龄，而学历又会影响到孩子智力，因此会造成父母的生育年龄也会影响孩子智力的结果。

因此, 当两个控制变量之间没有关系时, 需要构建无交互作用的双因素方差模型, 当两个控制变量之间会相互影响时, 需要构建有交互作用的双因素方差模型。

此外, 还有一些其他与方差分析相关的模型和检验方法。比如仅对模型中某些观察变量感兴趣的协方差模型, 或其他与方差分析效果相同的非参数检验等。

5.2 单因素方差分析

单因素方差分析是最简单的方差分析模型, 本节讨论了单因素方差分析的模型思想和应用, 以及一些重要的统计量, 这些统计量也同样适用于方差分析中较复杂的情况。

5.2.1 单因素方差分析的数学思想与模型

单因素方差分析中仅有一个观测变量和一个控制变量。控制变量记录了每个观测值所隶属的水平。方差分析中一个水平就是一个总体, 也称为一个组, 尽管单因素方差只有两个变量, 但它涉及多个总体的分析。观测变量记录了观测值, 通常用 x_{ij} 表示第 i 个水平下的第 j 个观测值, 通过观测变量既可以算出全体数据的总均值, 也可以算出每一组的组内均值。

在单因素方差分析中, SST 、 SSE 、 SSA 是最重要的三个统计量, 其中 SST 又称为总离差平方和, 其计算公式为 $SST = \sum_{i=1}^k \sum_{j=1}^{n_i} (x_{ij} - \bar{x})^2$, 其中 $\bar{x} = \frac{1}{n} \sum_{i=1}^k \sum_{j=1}^{n_i} x_{ij}$, 用于计算全体数据与总均值的差的平方和, 度量了全体数据的分散程度; SSE 又称为组内离差平方和, 其计算公式为 $SSE = \sum_{i=1}^k \sum_{j=1}^{n_i} (x_{ij} - \bar{x}_i)^2$, 用于计算每个观测值与其所属组的组内均值的差的平方和, 度量了每个组内数据的分散程度; SSA 又称为组间离差平方和, 其计算公式为 $SSA = \sum_{i=1}^k n_i (\bar{x}_i - \bar{x})^2$, 其中 $\bar{x}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij}$, 用于计算各水平均值与总均值的差的平方和, 度量了不同组之间的分散程度。三个统计量满足 $SST = SSE + SSA$ 的关系。

方差分析假设每个总体都服从正态分布, 由于正态分布仅由均值和方差这两个参数决定, 且方差分析假设不同总体具有相同方差, 因此 SST 、 SSE 、 SSA 都使用均值来构造统计量, 不同总体的均值就能够代表不同总体的分布, 检验不同总体间是否具有差异也就是检验不同总体的均值是否具有差异。

方差分析想要观察的是, 当控制变量水平不同时, 观测变量是否也会不同, 即组内均值是否随组别的变化而变化。 SSA 度量了不同组的组内均值的波动程度, 不同组的组内均值波动越剧烈, SSA 就越大, 越有理由认为不同组存在差异; 反之, 就认为不同组没有差异。

但当不同组的组内样本分散程度较大时, 组内均值波动程度也会较大, 即 SSE 会影响 SSA 的大小, 当 SSA 较大时, 可能是组间存在较大的差异, 也可能是 SSE 较大。为

了消去 SSE 对 SSA 的影响，方差分析又构造了 F 统计量，其计算公式为 $F = \frac{SSA/(k-1)}{SSE/(n-k)}$ ，服从 $(k-1, n-1)$ 的 F 分布。当 F 值显著大于 1 时，就认为组间存在差异。 F 值是否显著大于 1 可从临界值表中查得。

5.2.2 检验样本是否满足方差分析的假设条件

在正式进行方差分析前，检验样本是否满足方差分析的假设条件是非常必要的一项工作。本节使用 R 的基本数据集中的 **Orange** 数据集作为案例数据。

```
> head(Orange)
  Tree age circumference
1    1 118             30
2    1 484             58
3    1 664             87
4    1 1004            115
5    1 1231            120
6    1 1372            142
```

Orange 中存放了 5 棵橘子树在 7 个年份的树围围长。使用 `head()` 函数查看 **Orange** 的前 6 行数据，显然，**Orange** 中存放了三个变量，其中，**Tree** 变量存放树的标号，**age** 变量存放观测的年份，这两个变量都是控制变量；**circumference** 变量存放树的围长，这是观测变量。**Orange** 中一共存放了 35 行数据。这种将控制变量和观测变量分开存放的形式十分方便，同时也允许各水平的观测数目不同而不会妨碍数据框的生成。

```
> library(reshape)
> cast(Orange, Tree~age, value="circumference")
  Tree 118 484 664 1004 1231 1372 1582
1     3  30  51  75  108  115  139  140
2     1  30  58  87  115  120  142  145
3     5  30  49  81  125  142  174  177
4     2  33  69 111  156  172  203  203
5     4  32  62 112  167  179  209  214
```

通过表格形式能够更清楚地观察 **Orange** 中的数据结构。`reshape` 程序包提供了用于转换数据格式的 `cast()` 函数，上述代码首先加载了 `reshape` 包，然后调用了 `cast()` 函数。`cast()` 函数中一共有三个参数，第一个参数指定 **Orange** 为被应用的数据集；第二个参数指定 **Tree** 变量为表格的行，**age** 变量为表格的列；第三个参数 `value` 则指定使用 **circumference** 的值填充表格。

由于未给 `cast()` 函数指定赋值对象，R 立即返回了一个 5 行 8 列的矩阵，这个矩阵中的第 1 列标出了该列的数据属于哪棵树，后 7 列则代表了 7 个不同年份的数据。**Tree** 中存储的“1”、“2”、“3”、“4”、“5”并不是数值型的，因此，在 `cast()` 的返回结果中 **Tree** 并未按照数字大小对行进行排列。

除去第 1 列后，这个 5 行 7 列的矩阵明确地标出了每个观测值对应的年份和树木标号。

比如，位于第 1 行第 1 列的数据对应的是第 3 棵树在第 118 年时的树围，位于第 2 行第 5 列的数据对应的则是第 1 棵树在第 1580 年的树围。由于 `Orange` 中每一水平所拥有的观测值数目都是一致的，因此 `cast()` 返回的矩阵是非常规整的一个矩阵。当数据集中每一水平对应的观测值数目不一致时，`cast()` 返回的矩阵中将出现空缺值。

```
> attach(Orange)
> bartlett.test(circumference, Tree)

Bartlett test of homogeneity of variances

data:  circumference and Tree
Bartlett's K-squared = 2.4607, df = 4, p-value = 0.6517
```

`Orange` 中有两个控制变量，我们不妨研究当控制了 `Tree` 时，`circumference` 是否会出现明显差异，即不同树的树围是否有差别。在正式进行方差分析前，还需首先检验 `Orange` 是否满足方差分析的两个假设条件，即每个组都服从正态分布及不同组的方差相同。

上述代码验证了 `Tree` 的 5 个水平的方差是否一致。为了便于后续调用函数，`attach()` 首先绑定了 `Orange` 数据集，`bartlett.test()` 函数则对数据集应用了 Bartlett 检验。在 `bartlett.test()` 函数中指定了两个参数，第一个参数指定被检验的对象，第二个参数指定被检验对象的分组方法，这两个参数不能任意互换位置，否则将会生成很奇怪的结果。

R 依次返回了 Bartlett 检验的卡方值、自由度和 p 值，由于 `Tree` 一共有 5 个水平，因此自由度为 4，而 p 值为 0.6517，高于 0.05，因此认为在不同水平下 `circumference` 的方差是相同的。

```
> for (i in unique(Tree)){
+   T <- subset(Orange, Tree==i)
+   R <- shapiro.test(T$circumference)
+   print(R)
+ }

Shapiro-Wilk normality test

data:  T$circumference
W = 0.921, p-value = 0.4768

Shapiro-Wilk normality test

data:  T$circumference
W = 0.9119, p-value = 0.4094

Shapiro-Wilk normality test

data:  T$circumference
W = 0.9177, p-value = 0.4514
```

```

Shapiro-Wilk normality test

data:  T$circumference
W = 0.9036, p-value = 0.3531

Shapiro-Wilk normality test

data:  T$circumference
W = 0.9173, p-value = 0.4484

```

有关数据是否服从正态分布的检验有 `shapiro.test()` 函数和 `ks.test()` 函数，上述代码使用 `shapiro.test()` 函数检验了 `Orange` 中 5 个总体的观测值是否都服从正态分布。由于 `shapiro.test()` 函数仅能逐一检验 5 个总体，因此，上述代码使用 `for` 循环实现了逐一检验的功能。

`for` 循环的第 1 行代码使用 `unique()` 函数取出了 `Tree` 中所有不重复的数字，也就是 1~5，并令 i 从 1 到 5 循环一遍。这句代码中“`i in unique(Tree)`”与“`i in 1:5`”具有相同的作用。第 2 行代码使用 `subset()` 函数抽出了 `Orange` 中 `Tree` 值为 i 的子集，注意，它的第二个参数是“`Tree==i`”，如果写为“`Tree=i`”就会出错。

`for` 循环的第 3 行代码检验了 T 中 `circumference` 是否服从正态分布，最后一行代码则在屏幕上打出了 R 的结果。显然，由于每次循环中 i 值都会改变，因此子集 T 中的数据也会改变，结果 R 也会有所不同。循环执行完毕后，R 返回了 5 次 Shapiro-Wilk 检验的结果。这 5 次检验的 p 值都大于 0.05，因此，五个总体的观测值都服从正态分布。

5.2.3 构建单因素方差分析模型

经过检验，`Orange` 数据集中的控制变量 `Tree` 和观测变量 `circumference` 满足方差分析的两条重要假设，因此，可以对这两个变量做方差分析。R 提供了构建方差分析模型的函数 `aov()`，我们可以很方便地调用这个函数。

```

> aov(circumference~Tree)
Call:
aov(formula = circumference ~ Tree)

Terms:
              Tree Residuals
Sum of Squares  11840.86 100525.43
Deg. of Freedom      4       30

Residual standard error: 57.88651
Estimated effects are balanced

```

上述代码在 `aov()` 函数中仅指定了一个参数，这个参数以“观测变量~控制变量”

的形式指定了方差分析的对象。显然，circumference 和 Tree 的位置是不能随意互换的。由于在上文中已经 attach 了 Orange 数据框，因此此时可以直接输入 Orange 中的变量名，否则 R 将报错。此时可以用 \$ 将数据框和变量名连接起来，写成 “aov(Orange\$circumference~Orange\$Tree)” 这种形式。此外，aov() 也提供了参数 data 用于简化书写，如 “aov(circumference~Tree,Orange)” 的形式也是合法的，其中的 “Orange” 就是 “data=Orange” 的简写。

观察 R 返回的结果，Sum of Squares 一行显示该方差分析模型的组间离差平方和是 11 840.86，组内离差平方和是 100 525.43；Deg. of Freedom 一行显示 Tree 的自由度是 4，circumference 的自由度是 30，circumference 中虽然有 35 个数据，但由于此时它被分为了 5 组，因此此时自由度不再是 $35-1=34$ ，而变为了 $5 \times (7-1)=30$ 。R 同时也返回了 circumference 的标准误为 57.886 15。

```
> real.result <- aov(circumference~Tree)
> anova(real.result)
Analysis of Variance Table

Response: circumference
          Df Sum Sq Mean Sq F value Pr(>F)
Tree         4  11841   2960.2    0.8834  0.4857
Residuals   30 100525   3350.8
```

虽然 aov() 返回了不少信息，但这还不是方差分析模型所能返回的全部信息。anova() 函数和 summary() 函数都能查看方差分析模型的全部信息，上述代码首先将 aov() 函数的分析结果赋给了 real.result 变量，然后用 anova() 函数查看了其中的信息。

在 R 返回的表格中，Df 是两个变量的自由度，Sum Sq 一列是模型的组间离差平方和与组内离差平方和，Mean Sq 一列是模型的组间方差和组内方差，F 和 value Pr(>F) 则分别给出了模型的 F 统计量和 p 值。这个模型的 p 值大于 0.04，因此认为不同水平的观测值是没有差异的，也就是不同橘子树的树围并无不同。

```
> plot(circumference~Tree)
```

单因素方差模型的结果虽然表明 5 棵橘子树的树围并无差别，但仍不妨尝试用其他的方法佐证这一观点。箱线图是同时观察不同总体的均值的最佳图形，上述代码在 plot() 函数中指定了 circumference~Tree 参数，表示以 Tree 为水平，以 circumference 为观测值绘制图形，这两个变量的位置不可任意调换。观察图 5.1，plot() 函数返回了 5 个箱线图，分别代表 5 个不同的橘子树。此时这句命令所能达到的效果与命令 “boxplot(circumference~Tree)” 是一样的。由该图可知，这 5 棵橘子树的均值均落在 100~170，5 棵橘子树的 “箱子” 大小近乎一致。

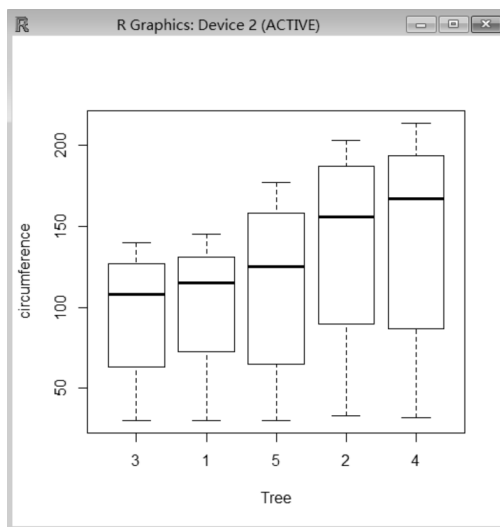


图 5.1 5 棵橘子树的树围数据箱线图

```
> pairwise.t.test(circumference, Tree)

Pairwise comparisons using t tests with pooled SD

data:  circumference and Tree

   3  1  5  2
1 1 - - -
5 1 1 - -
2 1 1 1 -
4 1 1 1 1

P value adjustment method: holm
```

除绘图外，多重 t 检验是另一种逐一检查方差分析中不同水平之间是否存在差异的方法。R 函数提供了 `pairwise.t.test()` 函数用于进行多重 t 检验。上述代码中 `pairwise.t.test()` 函数指定 `circumference` 为被检验对象，`Tree` 为控制变量。R 的返回结果给出了一个 t 检验的 p 值矩阵，矩阵的行和列标出了 t 检验的两个总体。

比如在上述例子中，矩阵的第 1 行、第 1 列的值代表第 1 棵树与第 3 棵树的 t 检验的 p 值，第 2 行、第 2 列则代表第 5 棵树与第 1 棵树的 t 检验的 p 值。由于 X 和 Y 进行 t 检验的结果与 Y 和 X 进行 t 检验的结果相同，因此，`pairwise.t.test()` 函数仅返回了一个 4 行 4 列的下三角矩阵。有关 `circumference` 的 t 检验 p 值矩阵全为 1，说明不同橘子树的树围是明显相同的。

R 的返回结果的最后一行说明 p 值修正方法为 `holm`，即没有修正。`pairwise.t.test()` 函数提供了 `p.adjust.method` 参数用于指定 p 值的修正方法，它的输入格式为 “`p.adjust.method="bonferroni"`”，可供选择的 p 值修正方法有 “`hochberg`”、“`hommel`”、“`bonferroni`”、“`BH`”、“`BY`”、“`fdr`”、“`none`” 等。

5.3 多因素方差分析

本节将单因素方差分析的模型思想推广到多因素方差分析中，与单因素方差分析相比，多因素方差分析涉及更多的统计量与更复杂的模型。本节将详细讨论如何用 R 构建多因素方差分析模型，以及如何解释模型的分析结果。

5.3.1 多因素方差分析的数学思想与模型

多因素方差分析指涉及多个控制变量和一个观测变量的方差分析模型，它的模型假设与单因素方差分析一致，也要求不同水平下的观测变量服从正态分布，并且方差相同。由于正态分布仅有均值和方差两个参数，而不同水平观测值的方差又相同，因此，检验多个控制变量的水平不同是否会导致观测值产生差异就等价于检验不同水平下观测变量的均值是否相同。这一基本思想与单因素方差分析同样保持一致。

双因素方差分析是多因素方差分析中最简单的情况，不妨先介绍双因素方差分析的模型构造方法，再推广到更复杂的情况中。双因素方差分析同时研究两个控制变量和一个观测变量之间的关系，一共涉及 5 个统计量，可构造出等式 $SST=SSA+SSB+SSAB+SSE$ 。

其中， SST 、 SSE 与单因素方差分析中的 SST 、 SSE 意义相同，分别代表方差分析中全体数据的总离差及各组的组内离差； SSA 和 SSB 分别代表控制变量 A 和控制变量 B 的组间离差； $SSAB$ 则代表两个控制变量的交互效应引起的离差，交互效应可理解为两个控制变量相互作用产生的混合影响，显然，多个控制变量可能会互相影响，也可能不会互相影响。

由于双因素方差分析同时涉及三个变量，此时方差分析模型中统计量的计算方法也较为复杂。不妨设控制变量 A 有 p 个水平，控制变量 B 有 q 个水平，每个交叉分组下有 r 个观测值，则总的离差平方和有计算公式 $SSE = \sum_{i=1}^p \sum_{j=1}^q \sum_{k=1}^r (x_{ijk} - \bar{x})^2$ ，其中 x_{ijk} 为 A 变量的第 i 个水平和 B 变量的第 j 个水平的交叉分组下的第 k 个观测值， \bar{x} 表示所有数据的均值；

SSA 和 SSB 的计算公式分别为 $SSA = qr \sum_{i=1}^p (\bar{x}_i^A - \bar{x})^2$ 及 $SSB = pr \sum_{j=1}^q (\bar{x}_j^B - \bar{x})^2$ ，

其中， \bar{x}_i^A 为因素 A 第 i 个水平下观测变量的样本均值， \bar{x}_j^B 为因素 B 第 j 个水平下观测变量的样本均值。 $SSA+SSB$ 度量了两个控制变量对观测变量的独立影响，合称为双因素方差分析的主效应。

由于等式 $SST=SSA+SSB+SSAB+SSE$ 恒成立，因此 $SSAB$ 可由 $SST-SSA-SSB-SSE$ 间接计算得出，而 SSE 的计算公式为 $SSE = \sum_{i=1}^p \sum_{j=1}^q \sum_{k=1}^r (x_{ijk} - \bar{x}_{ij}^{AB})^2$ ，其中， \bar{x}_{ij}^{AB} 是因素 A 、 B 在水平 i 、 j 下的观测变量样本均值， SSE 实际上度量的是方差分析模型中随机误差引起的数据波动，因此它又称为剩余效应。

有了统计量后，还需构造 F 值用以消除 SSE 对 SSA 、 SSB 和 $SSAB$ 的影响。双因素方

差分析需要计算三个 F 值，它们分别为 F_A 、 F_B 和 F_{AB} ，其中， $F_A = \frac{SSA/(p-1)}{SSE/pq(r-1)}$ ，它服从参数为 $(p-1, pq(r-1))$ 的 F 分布； $F_B = \frac{SSB/(q-1)}{SSE/pq(r-1)}$ ，它服从参数为 $(q-1, pq(r-1))$ 的 F 分布； $F_{AB} = \frac{SSAB/(p-1)(q-1)}{SSE/pq(r-1)}$ ，它服从参数为 $((p-1)(q-1), pq(r-1))$ 的 F 分布。将 F_A 、 F_B 和 F_{AB} 分别与临界值表相比较，即可得到三个 p 值。

控制变量多于两个时，模型的总离差同样可分解为主效应、交互效应、剩余效应的和。比如，当控制变量为三个时， SST 可分解为 $SSA+SSB+SSC+SSAB+SSAC+SSBC+SSABC+SSE$ ，其中， SSA 、 SSB 、 SSC 为三个控制变量对观测变量的独立影响； $SSAB$ 、 $SSAC$ 、 $SSBC$ 为三个控制变量两两交互后对观测变量的影响； $SSABC$ 则为三个控制变量一起交互后对控制变量的影响，这些变量的计算方法都与双因素方差分析类似，同样的，这三个控制变量可能会、也可能不会产生交互影响。

5.3.2 不考虑交互作用的双因素方差分析

本书仍以 Orange 数据集为分析对象。在 5.2 节中，`cast()` 函数已经告诉我们 Orange 中有两个控制变量 `Tree`、`age` 和一个观测变量 `circumference`，能够对其构建双因素方差分析模型。`Tree` 中有 5 个水平，`age` 中有 7 个水平，因此 Orange 能够分出 35 个交叉分组，每个交叉分组都仅有 1 个观测值。在其他的例子中交叉分组未必仅有 1 个观测值，当交叉分组中的观测值数目多于 1 时，`cast()` 函数将只能给出每个交叉分组中观测值的个数，而不能给出全体数据。

```
> attach(Orange)
> age <- as.factor(age)
> result <- aov(circumference~Tree+age)
> anova(result)
Analysis of Variance Table

Response: circumference
          Df Sum Sq Mean Sq F value    Pr(>F)
Tree       4  11841   2960.2   15.877 1.759e-06 ***
age        6   96051  16008.4   85.860 5.064e-15 ***
Residuals 24    4475    186.4
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

由于控制变量必须是因子型的数据，因此上述代码在绑定 Orange 数据框后，首先使用 `factor()` 函数将 `age` 变量转换为因子变量，然后利用 `aov()` 函数构建方差分析模型。如果忽略类型转换这一步骤，`aov()` 将无法正确地分组，也就无法生成正确的结果。

在 `aov()` 函数中仍以“观测变量 + 控制变量”的格式设置了参数，不过此时控制变量写为“`Tree+age`”，即根据两个控制变量构建方差分析模型。此时并未考虑控制变量的

交互作用，因此， $SST=SSA+SSB+SSE$ 。

仍旧使用 `anova()` 函数查看 `aov()` 函数生成的模型。由 R 的返回结果可知，控制变量 `Tree` 和 `age` 的自由度分别为 4 和 6，两个组间离差平方和分别为 11 841 和 96 051，组内离差平方和为 4 475，两个组间方差和组内方差分别为 2 960.2、160 008.4 和 186.4。`age` 的组间离差平方和与组间方差要比 `Tree` 的大很多，说明按照 `age` 分组后，不同组的数据均值波动要剧烈许多。

F_{Tree} 和 F_{age} 的值分别为 15.877 和 85.860，根据 F 值能够查得 p 值的大小，也就是 `Tree` 和 `age` 水平不同时，观测值相同的概率。`Tree` 和 `age` 的 p 值都小于 0.05，因此，`Tree` 或 `age` 水平不同时，`circumference` 的值也会有显著差异。此时两个 p 值都有一个扩展名“***”，返回结果的最后一行说明了它的意义。“***”表示接受原假设的概率无限趋近于 0；“**”表示接受原假设的概率小于 0.001；“*”表示接受原假设的概率小于 0.01，上述三个符号都表示强烈拒绝原假设；“.”表示接受原假设的概率小于 0.05；当接受原假设的概率大于 0.05 时，则不加扩展名符号，此时接受原假设。

对比上述结果与 5.2.3 节中 `Tree` 变量的统计量，构建单因素方差分析模型和无交互作用的双因素方差分析模型时，`Tree` 变量的组间离差平方和与组内方差并未发生变化，但 p 值却发生了极大的改变，显然，`age` 变量强烈地影响了 `Tree` 变量，二者很可能具有交互作用。

5.3.3 考虑交互作用的双因素方差分析

有交互作用的双因素方差分析同样使用 `aov()` 函数构建模型，只需在控制变量的位置添加 `Tree:age` 表示在模型中加入交互作用即可。

```
> result2 <- aov(circumference~Tree+age+Tree:age)
> anova(result2)
Analysis of Variance Table

Response: circumference
          Df Sum Sq Mean Sq F value Pr(>F)
Tree       4  11841   2960.2      15.877 0.000111 ***
age        6  96051  16008.4     85.860 1.16e-16 ***
Tree:age   24   4475    186.4       1.000 0.459118 .
Residuals  0         0
警告信息:
In anova.lm(result2) : 对几乎是完全拟合进行ANOVA F-检验的结果不会可靠
```

上述代码构建了考虑 `Tree` 和 `age` 的交互作用后的模型，此时 $SST=SSA+SSB+SSAB+SSE$ 。冒号表示交互作用，类似地，“ $Y \sim A+B+C+A:B+A:C+B:C+A:B:C$ ”表示以 Y 为观测变量，以 A 、 B 、 C 为控制变量构建考虑两两交互与三个变量相互交互后的三因素方差模型，“ $Y \sim A*B*C$ ”是它的简写格式；“ $Y \sim (A+B+C)^2$ ”则等价于 “ $Y \sim A+B+C+A:B+A:C+B:C$ ”，此时只考虑两两交互，抛去了 $A:B:C$ 部分。

使用 `anova()` 函数查看 `aov()` 函数的结果，R 给出了一个警告信息，提示此时的方

差检验不可靠。这是由于每个交叉分组中仅有一个观测值，因此，对 `Tree:age` 的统计量的计算将不可靠。由于出现了警告信息，此时并未给出 F 值和 p 值，在正常情况下，`Tree:age` 同样会给出一个 p 值，当 p 值大于 0.05 时，就要从方差模型中删除该项。在三因素方差分析中，只保留 $A:B$ 、 $A:C$ 、 $B:C$ 和 $A:B:C$ 中 p 值小于 0.05 的项。

`aov()` 函数中控制变量的输入顺序会对模型结果产生影响。即 `aov(Y~A+B+A:B)` 与 `aov(Y~B+A+A:B)` 产生的结果并不一致。`aov()` 函数在构建方差分析模型时采用贯序法，即对 `aov(Y~A+B+A:B)` 来说，计算 SSA 时不作调整，计算 SSB 时根据 SSA 作出调整，计算 $SSAB$ 时则根据 SSA 和 SSB 同时作出调整。

尽管上述代码并未给出 `Tree` 与 `age` 的交互项是否可信的结论，我们仍能用其他方法检验 `Tree` 与 `age` 是否具有交互作用。`interaction.plot()` 函数能够绘出有关交互作用的图形，它的基本书写格式为“控制变量 A ，控制变量 B ，观测变量 Y ”，如下代码绘制了 `age` 和 `Tree` 的交互效应图，并规定图形的 `bty` 参数为“1”，以便于避免图标超出图形框；设置 `cex.axis` 参数为 0.9，以便于在 x 轴放下 `age` 的所有 7 个水平。

```
> interaction.plot(age, Tree, circumference, bty="1", cex.axis=0.9)
```

图 5.2 是使用 `interaction.plot()` 函数绘出的交互效应图，它反映了 `Tree` 与 `age` 的交互程度。当两个控制变量 A 、 B 之间不存在交互效应时， A 增大与否不会扰乱 B 中各水平对应观测值的变化程度，此时交互效应图中的线条是相互平行的，否则，图中的线条将会相交，相交的角度越大，交互效应就越明显。

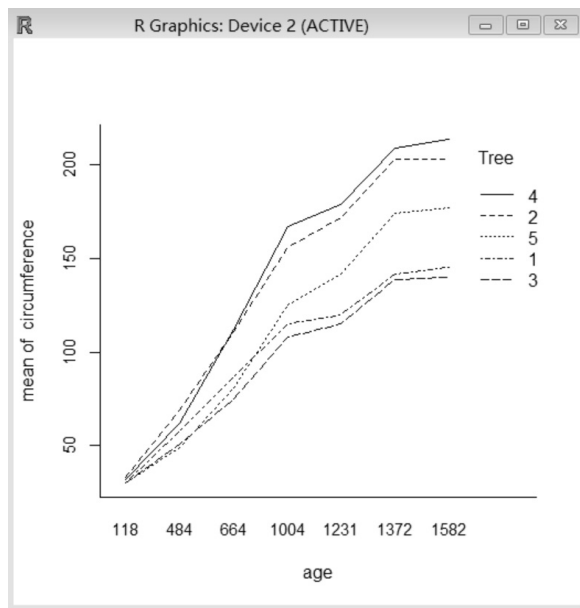


图 5.2 `Tree` 与 `age` 的交互效应图

观察图 5.2，图中 x 轴给出了 `age` 的 7 个水平，此时这 7 个年份仅代表 7 个水平，不区分数值大小，因此它们之间的距离是相等的。该图中一共画出了 5 条线，分别表示 `Tree` 的 5 个水平。这 5 条线的走势都呈上扬趋势，在 `age` 的前 4 个水平上略有交叉，后

三个水平则基本呈平行。总的来说, `Tree` 与 `age` 存在一定的交互效应, 但并不特别明显。

5.4 秩检验和协方差分析

本节的主题是方差分析模型中的最后两种情况, 秩检验和协方差分析。秩检验使用非参数的方法提供了控制变量不满足方差分析假设条件时的检验方法, 协方差分析则在模型中考虑了控制变量各水平不可控的情况。

5.4.1 对控制变量应用秩检验方法

在对 `Orange` 数据集构建双因素方差分析模型时, 我们发现控制变量 `age` 和 `Tree` 之间的交互效应并未特别显著, 但 `age` 却强烈地影响了 `Tree` 的可信程度。这可能是因为 `Orange` 中数据量太少, 也可能是因为变量 `age` 并不适合放入双因素方差分析模型中。不妨对 `age` 变量进行检验, 观察它是否满足方差分析的假设条件。

```
> attach(Orange)
> age <- as.factor(age)
> bartlett.test(circumference, age)

Bartlett test of homogeneity of variances

data:  circumference and age
Bartlett's K-squared = 24.5835, df = 6, p-value = 0.0004077
```

上述代码绑定了 `Orange` 数据框, 并将 `age` 变量转换为因子类型。`bartlett.test()` 函数检验了控制变量 `age` 的不同水平是否具有同一方差。在 R 的返回结果中 p 值小于 0.001, 因此拒绝原假设, `age` 不满足方差同一性假设条件。

由于 `age` 各水平的方差不同, 即便 `age` 的各水平都服从正态分布, 也不能仅使用均值来衡量各水平之间是否存在差异性, 因此, `age` 变量是不能被引入方差分析模型的, 这也是 5.3.3 节出现不合理结果的原因。此时需考虑使用不对数据分布作出要求的非参数检验方法。

与方差分析相关的非参数检验主要有 Kruskal-Wallis 秩检验和 Friedman 秩检验。Kruskal-Wallis 秩检验是 Wilcoxon 方法的一个推广, Wilcoxon 方法用于非正态总体的两样本均值检验, Kruskal-Wallis 秩检验则用于非正态总体的多样本均值检验。

```
> kruskal.test(circumference, age)

Kruskal-Wallis rank sum test

data:  circumference and age
Kruskal-Wallis chi-squared = 28.9341, df = 6, p-value = 6.261e-05
```

R 提供了 `kruskal.test()` 函数用于 Kruskal-Wallis 秩检验, 在 `kruskal.test()` 函数中只

需按照顺序指定观测变量和控制变量即可。R 返回了卡方值、自由度和 p 值， p 值仍是一个极小的值，因此，age 的不同水平下 circumference 的值有明显的差异。实际上橘子树的树围随着年份的增长显然会有一个明显的增长，因此控制变量 age 必然会影响 circumference 的值。

Friedman 秩检验同样利用秩统计量检验不同水平之间是否存在差异。R 提供了 `friedman.test()` 函数用于执行 Friedman 秩检验，但 `friedman.test()` 函数所应用的数据格式为向量或矩阵，因此，在正式执行 Friedman 秩检验前，还需将 circumference 数据转换为矩阵。

```
> library(reshape)
> c <- cast(Orange, Tree~age, value="circumference")
> d <- as.matrix(c)
> d
      118 484 664 1004 1231 1372 1582
3    30  51  75  108  115  139  140
1    30  58  87  115  120  142  145
5    30  49  81  125  142  174  177
2    33  69 111  156  172  203  203
4    32  62 112  167  179  209  214
> friedman.test(d)

      Friedman rank sum test

data:  d
Friedman chi-squared = 29.914, df = 6, p-value = 4.082e-05
```

上述代码首先利用 `reshape` 包中的 `cast()` 函数将 `Orange` 数据框转换为一个 `Tree` 为行、`age` 为列、`circumference` 为值的表格，并赋给了 `c`，然后使用 `as.matrix()` 函数将 `c` 转换为矩阵 `d`。上述代码查看了 `d` 中存储的数据，这种矩阵形式是 `friedman.test()` 函数所需要的格式。

最后一行代码对矩阵 `d` 应用了 `friedman.test()` 函数，R 同样返回了卡方值、自由度和 p 值，这个 p 值同样是一个非常小的数，Friedman 秩检验同样拒绝原假设，认为 age 处于不同水平时，circumference 值有明显的差异。

与方差分析模型相同，秩检验只能给出控制变量不同水平下的观测值是否具有差异，而不能指出具体是哪些水平之间具有差异，以及不同水平之间的差异有多大。对控制变量的不同水平成对地做两样本非参数检验能够回答上述问题，直接比较不同水平下观测变量的均值大小也能够大略地看出不同水平之间的差异。

5.4.2 协方差分析的假设与应用

协方差分析是方差分析中较为特殊的一类模型。有时观测变量不仅受到控制变量的影响，也受到其他数值型变量的影响。当影响观测变量的数值型变量不能被简单地划分为因子型变量时，就称这种数值型变量为协变量。

以橘子树为例，影响树的树围的变量不仅有树的种类、年份等，也有树所在海拔高度、平均气温、每日日照时间等变量。以树木所在海拔高度为例，这就是一个不受人类控制的实验变量（假如被研究的橘子树都是千年古木），很难恰好在同一海拔高度上找出几株年份相同的橘子树，而海拔高度又确实会影响树的树围，因此这就是一个数值型的协变量。

遗憾的是，Orange 数据集并未提供橘子树所处的海拔高度，不妨将 age 变量看作协变量引入协方差分析模型中。由于 age 变量本身就是数值型变量，因此，直接将 age 变量写入 aov() 函数即可。（如果你已经在 R 控制台中输入 “age<-as.factor(age)” 命令，则需要重新载入一次 Orange 数据集，确保 age 变量确实是数值型。）

```
> result<-aov(circumference~age+Tree)
> anova(result)
Analysis of Variance Table

Response: circumference
          Df Sum Sq Mean Sq F value    Pr(>F)
age         1  93772   93772  402.639 < 2.2e-16 ***
Tree         4  11841    2960   12.711 4.289e-06 ***
Residuals   29   6754     233
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

上述代码在 aov() 函数中以“观测变量~协变量+控制变量”的格式构建了协方差分析模型，这一代码与构建双因素方差分析模型的代码十分相似。anova() 函数返回的协方差分析结果也与双因素方差分析有相同的返回值。

观察 anova() 函数的返回结果，与双因素方差分析模型相比，此时 age 的自由度已从 6 变为了 1，F 值和 p 值也有较大的变化，Tree 的 F 值和 p 值也随之改变。但 age 和 Tree 的 p 值仍然十分小，这说明协变量 age 对观测值的影响是显著的，抛去 age 的影响后，控制变量 Tree 对观测值的影响也是显著的。

第6章 R中的相关分析和回归分析

相关分析和回归分析是联系非常紧密的两种分析方法，本章将首先介绍几种常见的相关分析，然后在此基础上介绍线性回归分析中的最小二乘法、逐步回归、逻辑回归和广义线性回归、非线性回归等。通过阅读本章，读者将较为全面地掌握常见回归分析方法的思想 and 应用。

6.1 多种相关系数的度量和分析

本节将围绕相关分析展开，讨论简单相关系数、偏相关系数和典型相关系数等多种相关系数的度量方法和作用，并讨论相关矩阵的构建和检验，以及有关典型相关分析的知识。

6.1.1 简单相关系数的计算和检验

相关关系是一种广泛存在于社会科学中的现象。当一个变量的值改变时，另一个变量的值也随之等比例地改变，我们就称两个变量之间存在相关关系。比如哥哥身高较高，则弟弟身高也会较高；商店的客流量增多，销售额也会增多；冬天气温上升，下雪的概率就会减少。这些都是相关关系的例子。

衡量两个变量的相关关系的系数称为相关系数。最常用的相关系数是简单相关系数，

它的计算公式为 $r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}}$ 。它由皮尔逊提出，专门用于衡量两个数值型变量的相关程度。

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4          0.2  setosa
2          4.9         3.0          1.4          0.2  setosa
3          4.7         3.2          1.3          0.2  setosa
4          4.6         3.1          1.5          0.2  setosa
5          5.0         3.6          1.4          0.2  setosa
6          5.4         3.9          1.7          0.4  setosa

> cor(iris[-5], use="everything", method="pearson")
              Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length      1.0000000  -0.1175698    0.8717538    0.8179411
Sepal.Width      -0.1175698    1.0000000   -0.4284401   -0.3661259
```

Petal.Length	0.8717538	-0.4284401	1.0000000	0.9628654
Petal.Width	0.8179411	-0.3661259	0.9628654	1.0000000

R 提供了 `cor()` 函数用于计算相关系数。在 `iris` 数据集中一共存储了 5 个变量，其中，前 4 个变量是数值型变量，最后一个变量是分类变量。`head()` 函数查看了它的前 6 行代码。

上述代码中的 `cor()` 函数中一共指定了三个参数，第一个参数指定 `cor()` 函数应用的对象是 `iris` 中除第 5 列之外的变量，即前 4 个变量；参数 `use` 用于指定缺失值处理方法，`everything` 表示某变量存在缺失值时，不计算它和其他变量的相关系数，此外，还有表示删掉变量中的缺失值后再计算相关系数的 `complete.obs` 选项等；参数 `method` 指定相关系数的类型，`pearson` 表示计算简单相关系数，此外，还有用于计算等级相关系数的 `spearman` 和 `kendall`，我们计算的 4 个变量都是数值型，因此，计算简单相关系数是合适的，倘若要对第 5 个变量计算相关系数，则需将参数 `method` 设置为 `spearman` 或 `kendall`。

`cor()` 函数返回了一个对称的方阵，位于 i 行 j 列的数据就是第 i 个变量和第 j 个变量的简单相关系数。简单相关系数是一个处于 $-1 \sim 1$ 之间的数，当它为正时，表示两个变量为正相关；当它为负时，表示两个变量为负相关，它的绝对值越接近 1，表示相关关系越强烈。

观察 `cor()` 返回的方阵，显然，每个变量和自身都绝对相关，因此，方阵的对角线上的元素全为 1。变量 1 与变量 3、变量 1 与变量 4、变量 3 与变量 4 之间的相关系数都为大于 0.8 的正数，呈现很强的正相关性。而变量 2 和其他三个变量的相关系数不仅全为负，而且绝对值都小于 0.5。

```
> cor.test(iris[,1],iris[,3])

Pearson's product-moment correlation

data:  iris[, 1] and iris[, 3]
t = 21.646, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.8270363 0.9055080
sample estimates:
      cor 
0.8717538 
> cor.test(iris[,1],iris[,4])

Pearson's product-moment correlation

data:  iris[, 1] and iris[, 4]
t = 17.2965, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7568971 0.8648361
sample estimates:
      cor 
0.7568971
```

```
0.8179411
> cor.test(iris[,3],iris[,4])

Pearson's product-moment correlation

data: iris[, 3] and iris[, 4]
t = 43.3872, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9490525 0.9729853
sample estimates:
      cor
0.9628654
```

`cor()` 函数只给出了两个变量相关程度的大小，并未给出两个变量相关系数的可信程度。`cor.test()` 函数提供了检验相关系数的功能。通常我们只关心相关程度强的那些变量对，对于相关程度不强的变量对来说，即便它们的相关关系十分可靠也不具备太大的意义。

上述代码分别两两检验了 `iris` 中第 1、3、4 个变量之间相关关系的可靠程度，在三行代码中，调用 `iris` 中变量时格式均为 “`iris[, 变量列数]`”，这代表抽出的是一个向量，假如去掉方括号中的逗号，抽出的就是一个数据框，`cor.test()` 函数将报错。

观察 R 的返回结果，三对变量的相关系数 p 值都小于 0.001，说明这三对变量之间具有非常明显的相关关系，当然，有时候也会出现两个变量的相关系数非常高，但 p 值却非常大，认为两个变量的相关关系不可靠的情况，这时无论相关系数有多高都是没有意义的。

6.1.2 散布矩阵图和偏相关系数

相关分析关心的是如何用相关系数度量不同变量间的关系，并为它们分类。散点图是反映两个变量之间相关关系的最直观图形，利用 `pairs()` 函数能够非常方便地画出一个矩阵散点图来。

```
> pairs(iris[-5])
```

上述代码为 `iris` 中除第 5 列以外的 4 列变量构造了矩阵散点图，图 6.1 所示为利用 `pairs()` 函数绘制出的图形。这是一个 4 行 4 列的矩阵，每一个矩阵元素就是一幅散点图，一共有 12 幅图形。观察这些图形，这显然是一个中心对称的矩阵，因此，图中仅有 6 幅散点图蕴含有效信息。

矩阵的对角线元素给出了 4 个变量名称，它们分别对应这 12 幅图形的 x 轴、 y 轴变量。以第 2 行、第 1 列图形为例，它的 x 轴为 `Sepal.Length` 变量， y 轴为 `Sepal.Width` 变量；而位于第 3 行、第 1 列的图形则以 `Sepal.Length` 变量为 x 轴，以 `Petal.Length` 变量为 y 轴。

观察图 6.1 中的散点图，`Sepal.Length` 与 `Petal.Length`、`Sepal.Length` 与 `Petal.Width`、`Petal.Length` 与 `Petal.Width` 形成的散点图较明显地集中在一条向右上方上升的直线上，`Sepal.Length` 与 `Petal.Length`、`Petal.Length` 与 `Petal.Width` 的散点图中的数据趋势又比

Sepal.Length 与 Petal.Width 更为明显, 这与 `cor()` 函数的结论是一致的。

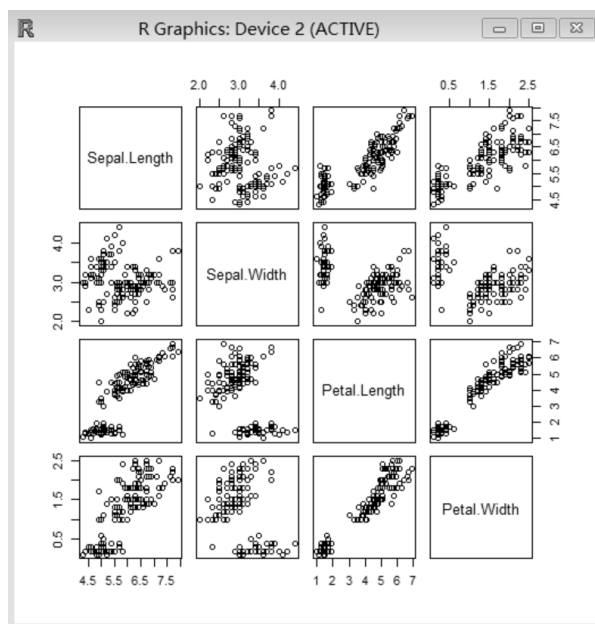


图 6.1 利用 `Pairs()` 函数绘制出的图形

在相关分析中, 我们不仅关心变量间的简单相关系数, 同样也关心变量间的偏相关系数。以 `iris` 数据集为例, 花萼的长度、花瓣的长度和花瓣的宽度这三者之间两两具有相关关系。但对花草植物有兴趣的读者都知道, 花瓣的长度同时会影响花萼的长度和花瓣的宽度, 花萼的长度和花瓣的宽度之间并不直接相关, 在 `iris` 数据集中 `Sepal.Length` 与 `Petal.Width` 可能是在 `Petal.Length` 的影响下才出现了相关关系。为了研究 `Sepal.Length` 与 `Petal.Width` 之间是否真的有相关关系, 我们需要考察它们的偏相关系数。

```
> library("corpcor")
> c<-cor(iris[-5],use="everything",method="pearson")
> cor2pcor(c)
      [,1]      [,2]      [,3]      [,4]
[1,] 1.0000000 0.6285707 0.7190656 -0.3396174
[2,] 0.6285707 1.0000000 -0.6152919 0.3526260
[3,] 0.7190656 -0.6152919 1.0000000 0.8707698
[4,] -0.3396174 0.3526260 0.8707698 1.0000000
```

`corpcor` 包提供了用于计算偏相关系数的 `cor2pcor()` 函数。上述代码加载了 `corpcor` 包, 并将 `iris` 数据集中前 4 个变量的简单相关系数矩阵赋给了 `c`, `cor2pcor()` 函数的应用对象是相关系数矩阵, 上述代码对 `c` 应用了 `cor2pcor()` 函数, 并返回了偏相关系数矩阵。

偏相关系数矩阵与简单相关系数矩阵类似, 也是一个对角线元素全为 1 的对称矩阵, 对比两个矩阵中的系数值, 在控制了第 3、4 个变量的影响后, 第 1、2 个变量的相关系数上升为 0.628; 在控制了第 1、4 个变量的影响后, 第 2、3 个变量的相关系数下降为 -0.61; 而控制了第 2、3 个变量的影响后, 第 1、4 个变量的相关系数下降为 -0.339, 直接从正

的强相关变为了负的弱相关。由此可见，简单相关系数和偏相关系数有较大的差异，简单相关系数为正，偏相关系数也可以为负。

```
> c<-cor(iris[,c(1,3,4)],use="everything",method="pearson")
> cor2pcor(c)
      [,1]      [,2]      [,3]
[1,]  1.0000000  0.5420163 -0.1620887
[2,]  0.5420163  1.0000000  0.8863158
[3,] -0.1620887  0.8863158  1.0000000
```

由于 Sepal.Length、Petal.Width 与 Petal.Length 两两之间的简单相关系数都非常高，不妨单独为它们计算一下偏相关系数，显然，此时 Sepal.Length 与 Petal.Width、Petal.Length 的相关系数都有不同程度的降低，特别是 Sepal.Length 与 Petal.Length 在控制了 Petal.Width 后，明显不再相关。显然，简单相关系数矩阵中关于这二者给出的相关关系是一种虚假相关。

6.1.3 典型相关分析

本节的最后一个知识点是典型相关分析，用于研究两组变量之间的相互关系。在自然界中存在两组随机变量相互关联的情况，这两组随机变量中的变量未必一一对应相关，但组与组之间存在相关关系，典型相关分析研究的就是整体的相关关系。

关于典型相关分析，R 提供了能够用于计算典型相关系数的 `cancor()` 函数。

```
> cancor(iris[,1:2],iris[,3:4])
$cor
[1] 0.9409690 0.1239369

$xccoef
      [,1]      [,2]
Sepal.Length -0.08757435  0.04749411
Sepal.Width   0.07004363  0.17582970

$ycoef
      [,1]      [,2]
Petal.Length -0.06956302 -0.1571867
Petal.Width   0.05683849  0.3940121

$xccenter
Sepal.Length Sepal.Width
    5.843333    3.057333

$ycenter
Petal.Length Petal.Width
    3.758000    1.199333
```

以 iris 数据集为例，不妨把花萼长度和花萼宽度看作一组变量，把花瓣长度和花瓣

宽度看作另一组变量,上述代码在 `cancor()` 函数的第一个参数处写入了 `iris` 中前两个变量,在 `cancor()` 函数的第二个参数处写入了 `iris` 中第三、四个变量,表示将这 4 个变量分为两组进行典型相关分析。

R 返回了一个由 4 个元素构成的列表。列表中的第一项是 `cor`, 它给出了两个典型相关系数。`cor` 中有两个值,说明 `cancor()` 函数共分析出两对相关变量,第一对变量的相关系数约为 0.940 9,这对变量的相关程度非常强,可以代表两组随机变量的绝大部分信息;第二对变量的相关系数不到 0.13,相关程度比较弱,能代表的信息也较少。

列表的第二个元素 `xcoef` 和第三个元素 `ycoef` 共同给出了相关变量的计算公式,第一对相关变量的计算公式为:

$$X_1 = -0.087\ 574\ 35 \cdot \text{Sepal.Length} + 0.070\ 043\ 63 \cdot \text{Sepal.Width}$$

$$X_1 = -0.069\ 563\ 02 \cdot \text{Petal.Length} + 0.056\ 838\ 49 \cdot \text{Petal.Width}$$

其中 X_1 是从第一组变量中抽出的相关变量, Y_1 是从第二组变量中抽出的相关变量。

第二对相关变量的计算公式为:

$$X_2 = 0.047\ 494\ 11 \cdot \text{Sepal.Length} + 0.175\ 829\ 70 \cdot \text{Sepal.Width}$$

$$Y_2 = -0.157\ 186\ 7 \cdot \text{Petal.Length} + 0.394\ 012\ 1 \cdot \text{Petal.Width}$$

X_2 是从第一组变量中抽出的相关变量, Y_2 是从第二组变量中抽出的相关变量。

列表的最后两个元素分别给出了 `Sepal.Length`、`Sepal.Width` 和 `Petal.Length`、`Petal.Width` 的均值。将各变量的均值与相关变量计算公式中各变量的系数相乘,可估计出每个元素在计算公式中的比重。

比如在第一对相关变量中, X_1 中 `Sepal.Length` 的系数与其均值相乘后约为 0.51, `Sepal.Width` 的系数与其均值相乘后约为 0.21, `Sepal.Width` 的比重显然比 `Sepal.Length` 小一些,这与 `Sepal.Length` 和 `Petal.Length`、`Petal.Width` 的相关程度更强有关;而 Y_1 中 `Petal.Length` 和 `Petal.Width` 计算得到的值分别为 0.26、0.06 也说明 `Petal.Length` 与第一组变量的联系要比 `Petal.Width` 更紧密。

将元素的系数与均值相乘毕竟还是有些麻烦,一个更方便的做法是,在做典型相关分析前将被分析的变量全部标准化,变量标准化后数据结构不会发生变化,只是变量中的数据按照比例进行了放缩,所有变量将一律服从参数为 (0, 1) 的正态分布。此时比较相关变量计算公式中每个元素的比重时直接比较元素的系数即可,这个技巧在后续内容中将会多次出现。

6.2 线性回归分析及其常规参数

线性回归分析是既经典又流行的一种统计分析方法,本节将介绍如何构造回归方程并对其进行修正。通过学习本节内容,读者将掌握线性回归分析的基本知识,学习本小节也是后续三个小节的准备工作。

6.2.1 对数据进行预处理

回归分析用于研究一个因变量和一组自变量之间的关系。当自变量只有一个时，需要构建一元线性回归模型；当自变量多于一个时，需要构建多元线性回归模型。线性回归模型希望找出一个确定的等式用于衡量因变量和自变量间的关系。

最简单的线性回归方程式可写为 $Y = a_0 + a_1X_1 + a_2X_2 + \dots$ 的形式，其中 Y 为因变量， X_1 和 X_2 分别为第一个自变量和第二个自变量， a_i 则为对应的自变量的系数，特别的， a_0 为常数项。显然自变量和因变量间具有很强的相关关系，当自变量发生改变时，因变量会随之等比例地改变。但与相关分析不同的是，相关分析中两个相互相关的变量地位是平等的，而回归分析中自变量和因变量的地位却不平等，比如我们认为父亲的身高会影响儿子的身高，但不能认为儿子的身高会影响父亲的身高。

自变量的系数与自变量对因变量的影响大小有关，自变量与因变量越相关，回归方程中自变量的系数就越大。与典型回归分析相似，自变量的量纲大小同样会影响系数的大小。为了消除自变量的量纲对系数的影响，不妨将自变量和因变量全部标准化。

```
> SL.sc<-scale(iris[1])
> SW.sc<-scale(iris[2])
> PL.sc<-scale(iris[3])
> PW.sc<-scale(iris[4])
```

R 提供了 `scale()` 函数用于变量的标准化，上述代码标准化了 `iris` 数据集中的前 4 个变量，并分别赋给了 `SL.sc`、`SW.sc`、`PL.sc` 和 `PW.sc`。`scale()` 函数默认的数据标准化的计算公式为 $X_{i\text{-new}} = \frac{X_i - \bar{X}}{S_x}$ ，每个数据的标准化值为该数据减去变量均值后除以变量标准差的值。它按比例缩放放了变量中的数据，并不影响变量的数据结构。这种数据标准化方法又称为 `z-score` 标准化，其他标准化方法还有 `Min-Max` 标准化和 `Decimal Scaling` 小数定标标准化等。

```
> range(SL.sc);range(SW.sc);range(PL.sc);range(PW.sc)
[1] -1.863780  2.483699
[1] -2.425820  3.080455
[1] -1.562342  1.779869
[1] -1.442245  1.706379
```

上述代码使用 `range()` 函数查看了 `SL.sc`、`SW.sc`、`PL.sc`、`PW.sc` 中的数值范围，原始数据中并没有负值，但数据标准化后均出现了负值，显然，数据在标准化后分散在 0 周围。在 4 个标准化变量中 `SW.sc` 的范围最大，说明它的数据结构最分散。

标准化数据同样也给出了一个非常重要的范围，在我们将回归方程用于预测时，代入回归方程进行计算的自变量都应该是标准化后的自变量，而且自变量的值不能超出上述代码所给的范围，比如预测一组 `SW.sc`、`PL.sc`、`PW.sc` 分别为 -3、2、1.5 的值对应的 `SL.sc` 值就是没有意义的，因为 `SW.sc` 值和 `PL.sc` 值都超出了可预测的范围，这种规则在只有一个自变量的值超出范围时也同样适用。

6.2.2 构建第一个回归模型

构建回归分析模型时，我们希望找到一个确定的等式用于计算因变量的值。从几何意义上来理解，一元线性回归涉及一个自变量和一个因变量，这两个变量能够画出一个二维平面上的散点图，而一元线性回归方程所要做的就是找出一条直线，使散点图上的点与这条线的距离之和最小，此时 a_0 是直线与 y 轴的截距；二元线性回归方程所要做的就是要在三维立体空间中找到一个平面；自变量多于两个时，将不再具有几何意义。

无论是几元线性回归，我们都希望全体数据点与线性回归方程的距离之和越小越好，线性回归方程离数据点越近，线性回归方程的准确度就越高。在构建线性回归模型时，自变量和因变量的值都是已知的，求解线性回归方程也就相当于求解方程中每个自变量的系数值。

最小二乘法是应用最广泛的回归系数估计方法，它将因变量的真实值和估计值的差视为误差项，并用误差项的平方和作为统计量，其计算公式为 $S_a = \sum_{i=1}^n \left(y_i - \sum_{j=0}^p x_{ij} a_j \right)^2$ ，最小二乘法通过对 S_a 求偏导的方法分别求出每个 a_j 的值，使 S_a 达到最小，此时 a_j 就是第 j 个自变量的系数。

```
> lm.sc<-lm(SL.sc~SW.sc+PL.sc+PW.sc)
> summary(lm.sc)

Call:
lm(formula = SL.sc ~ SW.sc + PL.sc + PW.sc)

Residuals:
    Min       1Q   Median       3Q      Max
-1.00012 -0.26555  0.02264  0.23802  1.02129

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.176e-16  3.102e-02   0.000      1
SW.sc        3.426e-01  3.508e-02   9.765 < 2e-16 ***
PL.sc        1.512e+00  1.209e-01  12.502 < 2e-16 ***
PW.sc       -5.122e-01  1.174e-01  -4.363 2.41e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3799 on 146 degrees of freedom
Multiple R-squared:  0.8586,    Adjusted R-squared:  0.8557
F-statistic: 295.5 on 3 and 146 DF,  p-value: < 2.2e-16
```

`lm()` 函数能够很方便地使用最小二乘法计算出回归方程中的系数值，与方差分析相似，`lm()` 函数的输入格式为“因变量~自变量”。上述代码以 `SL.sc` 为因变量，以 `SW.sc`、`PL.sc`、`PW.sc` 为自变量构建了线性回归方程，`summary()` 函数查看了 `lm()` 函数的全部信息。

R 关于 `summary()` 函数的返回结果是一个列表，`Call` 元素给出了线性回归模型的基本信息；`Residuals` 元素给出了线性回归模型的残差的 5 个分位数，它们用于衡量回归方程的合理程度；`Coefficients` 给出了每一个自变量的系数和可信程度；剩下的一些行则是关于回归模型更详细的信息。

不妨倒着查看 R 的返回结果，返回结果的最后一行给出了模型总体的 F 值、自由度和 p 值。这个 p 值用于衡量方程总体的显著程度，该值小于 0.001，因此该回归方程是可信的。倒数第 2 行表明该回归方程的 R 方是 0.858 6，调整 R 方是 0.855 7，表明回归方程能够解释原始数据中约 85.57% 的信息，同样表明该回归方程的可信程度很高。总体 p 值和 R 方直接给出有关回归模型的整体信息，当 p 值大于 0.05，或 R 方小于 0.60 时，回归方程很可能引入了不恰当的自变量，或者原始数据不适合做线性回归分析。

总体 p 值和 R 方通过检验后，还需观察 `Coefficients` 给出的更详细的信息。`Coefficients` 中 `Estimate`、`Std. Error`、`t value`、`Pr(>|t|)` 分别给出了自变量的系数、标准误、 t 值和 p 值。`Intercept` 项给出的是有关常数项 a_0 的信息， a_0 的系数值非常小， p 值又大于 0.05，因此回归方程中不适合引入常数项；自变量 `SW.sc`、`PL.sc`、`PW.sc` 的 p 值都小于 0.05，因此通过了检验，应该留在回归方程中。

6.2.3 修正方程并检验残差

`lm()` 函数使用 1 代表常数项，只需在自变量部分加入 “-1”，`lm()` 函数将在方程模型中去掉常数项。如下代码以 `SL.sc` 为因变量，以 `SW.sc`、`PL.sc`、`PW.sc` 为自变量构建了一个不含常数项的回归模型。

```
> lm.sc<-lm(SL.sc~SW.sc+PL.sc+PW.sc-1)
> summary(lm.sc)

Call:
lm(formula = SL.sc ~ SW.sc + PL.sc + PW.sc - 1)

Residuals:
    Min       1Q   Median       3Q      Max
-1.00012 -0.26555  0.02264  0.23802  1.02129

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
SW.sc    0.34258     0.03496   9.799 < 2e-16 ***
PL.sc    1.51175     0.12050  12.545 < 2e-16 ***
PW.sc   -0.51224     0.11701  -4.378 2.26e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3786 on 147 degrees of freedom
Multiple R-squared:  0.8586,    Adjusted R-squared:  0.8557
F-statistic: 297.6 on 3 and 147 DF,  p-value: < 2.2e-16
```

查看 `summary()` 函数的返回结果，此时无论是方程总体的 p 值，还是每个自变量的 p 值都通过了检验，模型的 R 方也未发生变化。观察 `Coefficients` 提供的系数值，现在根据回归模型，可知回归方程 $SL.sc = 0.34258 \times SW.sc + 1.51175 \times PL.sc - 0.51224 \times PW.sc$ 成立。

显然，当 $PL.sc$ 、 $PW.sc$ 不变时， $SW.sc$ 每增加一个单位， $SL.sc$ 将增加约 0.34 个单位；当 $SW.sc$ 、 $PW.sc$ 不变时， $PL.sc$ 每增加一个单位， $SL.sc$ 将增加约 1.51 个单位；当 $SW.sc$ 、 $PL.sc$ 不变时， $PW.sc$ 每增加一个单位， $SL.sc$ 将减少约 0.51 个单位。 $PL.sc$ 对 $SL.sc$ 的影响最大，约为 $PW.sc$ 的 3 倍， $SW.sc$ 的 5 倍。

方程的 p 值通过检验后，仍不代表这是一个合理的模型，还需回到 `Residuals` 元素上，检验方程的残差是否合理。

```
> plot(lm.sc, 1)
```

利用 `plot()` 函数能绘制有关线性回归模型的好几种图形，上述代码指定为线性回归模型绘制第一种图形，返回结果如图 6.2 所示。

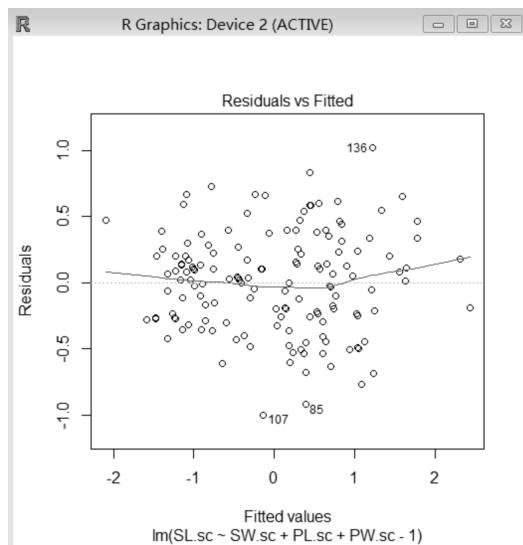


图 6.2 线性回归模型 `lm.sc` 的残差图

残差用于衡量每个观测值的真实值和估计值之间的误差项，首先，残差不应该太大。图 6.2 所示为线性回归模型 `lm.sc` 的残差图，图中每个观测值的残差都处于区间 -1 到 1 之间，说明这些点都紧紧围绕在方程周围，没有太大的误差。当残差过大，比如，残差的绝对值达到 3 时，对应的观测值就显然是一个异常值，应该剔除它后再做回归模型，图 6.2 标出了残差最大的三个观测值，不过它们并未过于异常，因此不做处理。

其次，残差应该围绕 0 均匀分布，并且不呈现出任何明显的规律。图 6.2 同样绘制了一条代表观测值点残差分布的红线，这条曲线基本紧挨着代表 0 的虚线，因此，残差值是围绕 0 均匀分布的。同时，图 6.2 中的点呈现散漫的分布，并未紧挨在一条直线或曲线上，因此回归模型 `lm.sc` 的残差检验同样是通过的。

```
> plot(lm.sc, 2)
```

上述代码指定为 `lm.sc` 模型绘制第二类图形, 也就是残差 QQ 图, R 的返回结果如图 6.3 所示。

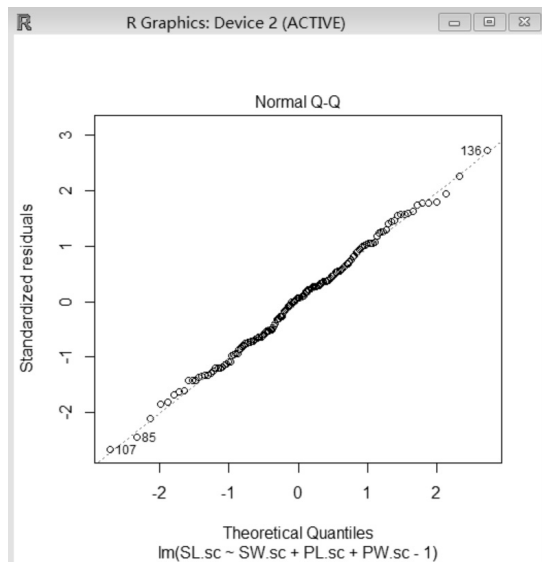


图 6.3 线性回归模型 `lm.sc` 的残差 QQ 图

与正态 QQ 图相似, 残差 QQ 图反映了观测值是否服从正态分布。观察图 6.3, 残差值紧紧围绕在斜穿图形的直线上, 说明观测值基本服从正态分布, 图中标出的三个异常值也紧紧地挨着这条斜线, 因此线性模型 `lm.sc` 的性质是相当好的。

除残差图和残差 QQ 图外, `plot()` 还能绘制有关线性回归模型的其他四五种图形, 只需修改参数值即可一一查看。

6.3 使用逐步回归筛选自变量

逐步回归是一种特殊的回归方程构建方法, 它采用专门的统计量衡量自变量在回归方程中的重要程度, 并依据该值对自变量做出筛选。本节将介绍逐步回归的思想和模型, 并展示如何在数据集上应用这种方法。

6.3.1 逐步回归的思想与分类

在拟合回归模型时, 我们既希望模型能完美地拟合出真实情况, 又希望模型计算起来不要过于复杂。因此, 我们总希望模型中包含最少的且最重要的自变量。当自变量较少时, 逐一实验不同的自变量组合是可行的, 但当自变量多于 5 个时, 自变量的组合将十分多, 从中挑选出最佳的变量组合也将十分困难。

逐步回归提供了一些用于比较方程拟合优度的统计量, 可以简化最佳变量组合的挑选过程。这些统计量从所有可能的回归方程中按照一定的准则挑选最优的方程, 最常用的统计量有修正复相关系数、预测平方和、 C_p 和 AIC 等。

复相关系数度量的是一个变量和一组变量之间的相关程度，显然，它能够度量因变量与一组自变量之间的相关程度。修正复相关系数在复相关系数的计算公式上添加了一个修正系数，它的计算公式为 $\sqrt{\frac{(SST-SSE)/p}{SST/n-1}}$ ，其中 p 为自变量的个数， n 为观测值的个数， SST 和 SSE 分别为总的离差平方和及残差平方和。当修正复相关系数达到最大时，回归方程是最优的，这一条件与 $\frac{SSE}{n-p-1}$ 达到最小等价。

预测平方和度量了 n 个观测因变量值的真实值和估计值的误差大小，它的计算公式为 $\sum_{i=1}^n (y_i - \hat{y}_i)^2$ ，它与用于计算系数值的最小二乘法十分相似，当预测平方和达到最小时，回归方程是最优的。

C_p 准则使用 SSE 度量回归方程的优度，其计算公式为 $C_p = \frac{SSE_p}{\frac{SSE}{n-p-1}(x_1, x_2 \cdots x_m)}$ ($n-2p-2$)，设共有 m 个自变量供挑选，则 SSE_p 为包含 p 个自变量的回归方程的残差平方和， $\frac{SSE}{n-p-1}(x_1, x_2 \cdots x_m)$ 为包含全部 m 个自变量的回归方程的均方残差。 C_p 准则要求 C_p 和 $|C_p - p|$ 都达到最小，此时回归方程是最优的。

AIC 准则是最后一种回归方程拟合优度的判断准则。对于含有 p 个自变量的回归方程，其计算公式为 $AIC = n \cdot h \left(\frac{SSE}{n-p-1} \right) + 2p + 2$ ，当它达到最小时，回归方程式是最优的。

在上述 4 种统计量中，预测平方和计算量较大， C_p 准则又较复杂，AIC 准则是最实用的自变量筛选方法，也是 R 默认的逐步回归模型计算方法。

6.3.2 构建逐步回归模型

仍以 SL.sc 为因变量，以 SW.sc、PL.sc、PW.sc 为自变量构建不含常数项的线性回归模型，并将模型结果赋给 lm.sc，不妨尝试从这三个自变量中进一步筛选出最佳的两个。

```
> step(lm.sc)
Start:  AIC=-288.44
SL.sc ~ SW.sc + PL.sc + PW.sc - 1

      Df Sum of Sq   RSS   AIC
<none>             21.067 -288.44
- PW.sc    1      2.7466  23.814 -272.06
- SW.sc    1     13.7602  34.827 -215.04
- PL.sc    1     22.5548  43.622 -181.26

Call:
lm(formula = SL.sc ~ SW.sc + PL.sc + PW.sc - 1)
```

Coefficients:

SW.sc	PL.sc	PW.sc
0.3426	1.5118	-0.5122

step() 函数提供了构建逐步回归模型的功能，它接受一个 lm 类型的输入对象，并默认使用 AIC 准则筛选输入对象中的自变量。上述代码在 lm.sc 上应用了 step() 函数。

R 返回了一个列表。返回结果的前两行分别给出了初始的 AIC 值和初始的自变量组合。然后给出了一个有关逐步回归的表格，表格包括 Df、Sum of Sq、RSS、AIC 四列，其中 AIC 一列表明不对自变量组合进行改变时，AIC 值为 -288.44；从自变量中删去 PW.sc 时，AIC 值上升为 -272.06；从自变量中删去 SW.sc 时，AIC 值上升为 -215.04；从自变量中删去 PL.sc 时，AIC 值上升为 -181.26。

显然，初始的自变量组合已经是最优的自变量组合，因此 step() 函数并未继续向后计算。如果删去一个自变量后方程的 AIC 值将变小，step() 会在删去该自变量的基础上进一步计算是否还有自变量能够在删去后再次减小方程的 AIC 值，直至 AIC 跌到最低点，不再随着自变量的删去而减小为止。

step() 函数同样提供了其他用于判断方程拟合优度的参数。Sum of Sq 一列给出了删除该变量后方程的残差平方和上升的幅度。显然，从自变量中删去 PW.sc 时，方程的残差平方和将上升 2.746 6；从自变量中删去 SW.sc 时，方程的残差平方和将上升 13.760 2；从自变量中删去 PL.sc 时，方程的残差平方和将上升 22.554 8。方程的残差平方和越小，方程的拟合优度就越好，因此倘若必须要删去一个变量，则删去 PW.sc 对方程的影响是最小的。

```
> lm.new<-lm(SL.sc~SW.sc+PL.sc-1)
> summary(lm.new)

Call:
lm(formula = SL.sc ~ SW.sc + PL.sc - 1)

Residuals:
    Min       1Q   Median       3Q      Max
-1.16125 -0.28366  0.00093  0.25907  0.94868

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
SW.sc    0.31346     0.03637   8.619 9.46e-15 ***
PL.sc    1.00605     0.03637  27.663 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4011 on 148 degrees of freedom
Multiple R-squared:  0.8402,    Adjusted R-squared:  0.838
F-statistic: 389 on 2 and 148 DF,  p-value: < 2.2e-16
```

上述代码构建了以 SL.sc 为因变量，以 SW.sc 和 PL.sc 为自变量的不含常数项的线性回归模型，使用 summary() 函数查看模型的具体结果，此时 PL.sc 对 SL.sc 的影响仍

约为 SW.sc 的 5 倍，但方程总体的 R 方降低了，说明删去一个自变量后，方程对原始数据中信息的解释变少了，同时 SW.sc 的 p 值增大了一些，它虽然仍是非常可信的，但与未删去自变量 PW.sc 时相比，它的可信程度下降了。

6.4 哑变量和逻辑回归

本节将在线性回归模型的基础上进一步讨论如何向模型中引入分类变量。分类变量的各个类别不能直接作出数值上的比较，本节的主题是如何将分类变量转换为哑变量，并演示如何向普通的线性模型中引入哑变量。

6.4.1 哑变量和逻辑回归的思想

分类变量是一种只包含几个特定值的变量，比如人的性别分为男女；婚姻状况分为未婚、一婚和二婚，那么记录性别的变量和记录婚姻状况的变量就是分类变量，显然，性别是一个二分类变量，婚姻状况是一个三分类变量。当记录分类变量时，既可以直接输入字符，也可以使用数值来代表每个类别，比如在性别变量中用 1 代表性别为男，用 2 代表性别为女等。

截至目前，关于回归分析的所有讨论都局限在数值型变量中，并未提到如何在回归模型中加入分类变量。首先，当分类变量是字符型时，由于无法计算字符的大小，因此回归模型中无法纳入分类变量；其次，当分类变量是数值型时，我们也要非常小心才行。

不妨以收入为因变量，以年龄和婚姻状况为自变量构建一个线性回归模型。按照线性模型来说，这个模型应该写为 $Y_{\text{收入}} = a_0 + a_1 \cdot X_{\text{年龄}} + a_2 \cdot X_{\text{婚姻}}$ ，当用数值来代表婚姻状况时，婚姻状况之间就有了大小关系。比如设 1 代表未婚，2 代表一婚，3 代表二婚。那么当婚姻状况确定时，一婚的收入一定比未婚多 a_2 ，而二婚又比未婚多两个 a_2 ，此时不同的婚姻状况之间就有了倍数关系。但实际上婚姻状况之间未必恰好都有同样的倍数关系，因此向模型中直接引入分类变量是不恰当的。

为了消除分类变量的类别之间相互的影响，最佳的做法是将每个类别都单独拆开。此时线性回归模型的公式即为 $Y_{\text{收入}} = a_0 + a_1 \cdot X_{\text{年龄}} + a_{21} \cdot X_{\text{未婚}} + a_{22} \cdot X_{\text{一婚}} + a_{23} \cdot X_{\text{二婚}}$ ，其中 $X_{\text{未婚}}$ 、 $X_{\text{一婚}}$ 、 $X_{\text{二婚}}$ 均为取值为 0、1 的变量，当观测值的婚姻状况为未婚时，其 $X_{\text{未婚}}$ 值即为 1， $X_{\text{一婚}}$ 值即为 0， $X_{\text{二婚}}$ 值即为 0；当观测值的婚姻状况为一婚时，其 $X_{\text{未婚}}$ 值即为 0， $X_{\text{一婚}}$ 值即为 1， $X_{\text{二婚}}$ 值即为 0；当观测值的婚姻状况为二婚时，其 $X_{\text{未婚}}$ 值即为 0， $X_{\text{一婚}}$ 值即为 0， $X_{\text{二婚}}$ 的值即为 1。

那么，当观测值的婚姻状况为未婚时，对应的线性回归模型为 $Y_{\text{收入}} = a_0 + a_1 \cdot X_{\text{年龄}} + a_{21}$ ；当观测值的婚姻状况为一婚时，对应的线性回归模型为 $Y_{\text{收入}} = a_0 + a_1 \cdot X_{\text{年龄}} + a_{22}$ ；当观测值的婚姻状况为二婚时，对应的线性回归模型为 $Y_{\text{收入}} = a_0 + a_1 \cdot X_{\text{年龄}} + a_{23}$ 。此时婚姻状况被拆为三个单独的变量，不同分类之间不再具有倍数关系，方程的合理性也得到了提升。

进一步简化线性模型，可以将 $X_{\text{未婚}}$ 变量从模型中去掉。那么当 $X_{\text{一婚}}$ 、 $X_{\text{二婚}}$ 取值均

为 0 时, 观测值为未婚, 对应的线性方程为 $Y_{\text{收入}} = a_0 + a_{21} + a_1 \cdot X_{\text{年龄}}$, 其中 $a_0 + a_{21}$ 为新方程的常数项; 当 $X_{\text{一婚}}$ 取值为 1, $X_{\text{二婚}}$ 取值为 0 时, 观测值为一婚, 对应的线性方程为 $Y_{\text{收入}} = a_0 + a_{21} + a_1 \cdot X_{\text{年龄}} + a_{22} - a_{21}$, 其中 $a_{22} - a_{21}$ 为 $X_{\text{一婚}}$ 的系数; 当 $X_{\text{一婚}}$ 取值为 0, $X_{\text{二婚}}$ 取值为 1 时, 观测值为二婚, 对应的线性方程为 $Y_{\text{收入}} = a_0 + a_{21} + a_1 \cdot X_{\text{年龄}} + a_{23} - a_{21}$, 其中 $a_{23} - a_{21}$ 为 $X_{\text{二婚}}$ 的系数。

与拆分婚姻状况类似, 有 n 个分类的分类变量都可以拆分为 $n-1$ 个取值为 0 和 1 的分类变量并纳入回归方程中, 这些取值为 0 和 1 的分类变量称为哑变量, 而纳入了哑变量的线性回归模型称为逻辑回归模型。

6.4.2 向线性回归模型中纳入哑变量

iris 数据集不仅提供了 4 个数值型变量, 还提供了一个分类变量 Species。不妨试一试如何将 Species 纳入线性回归模型中。

```
> attach(iris)
> summary(Species)
   setosa versicolor  virginica 
       50         50         50
```

上述代码首先绑定了 iris 数据集, 然后用 summary() 函数查看了 Species 变量的摘要。由 R 的返回结果可知, Species 中共有三个分类, 分别为 setosa、versicolor 和 virginica, 它们每一个都有 50 条记录, 加起来刚好对应 150 个观测值。

由于有三个分类, 因此将其转换为两个哑变量。

```
> Spe1<-rep(0,150);Spe2<-rep(0,150)
> for(i in which(Species=="versicolor"))
+   Spe1[i]<-1
> for(i in which(Species=="virginica"))
+   Spe2[i]<-1
```

上述代码首先创建了两个长度为 150、值全为 0 的向量 Spe1 和 Spe2, rep() 函数将 0 值重复了 150 次。接下来的两个 for 循环修改了 Spe1 和 Spe2 中的值。第一个 for 循环使用 which() 函数指定循环范围为 Species 的值为 versicolor 元素的下标, 并令 Spe1 中位于循环范围内的元素值均为 1, 此时 Spe1 即为一个哑变量, 当它取 1 时, 表示观测值为 versicolor 类型; 当它取 0 时, 表示观测值为其他类型; 类似地, Spe2 同样为一个哑变量, 当它取 1 时, 表示观测值为 virginica 类型; 当它取 0 时, 表示观测值为其他类型。

```
> SL.sc<-scale(iris[1])
> SW.sc<-scale(iris[2])
> PL.sc<-scale(iris[3])
> PW.sc<-scale(iris[4])
> lm.log<-lm(SL.sc~SW.sc+PL.sc+PW.sc+Spe1+Spe2)
> summary(lm.log)
```

Call:

```
lm(formula = SL.sc ~ SW.sc + PL.sc + PW.sc + Spe1 + Spe2)

Residuals:
    Min       1Q   Median       3Q      Max
-0.95915 -0.26416  0.01085  0.24460  0.88282

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.5327     0.1796  -2.967  0.00353 **
SW.sc         0.2610     0.0453   5.761 4.87e-08 ***
PL.sc        1.7678     0.1461  12.101 < 2e-16 ***
PW.sc       -0.2901     0.1392  -2.084  0.03889 *
Spe1         1.2360     0.4030   3.067  0.00258 **
Spe2         0.3622     0.1437   2.521  0.01280 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3705 on 144 degrees of freedom
Multiple R-squared:  0.8673,    Adjusted R-squared:  0.8627
F-statistic: 188.3 on 5 and 144 DF,  p-value: < 2.2e-16
```

准备好哑变量后，即可像构造普通的线性回归模型那样构造逻辑回归模型。上述代码首先对 4 个数值变量进行了标准化，然后以 SL.sc 为因变量，以 SW.sc、PL.sc、PW.sc 和 Spe1、Spe2 为自变量构建了带有截距项的线性回归模型。使用 summary() 函数查看模型结果，R 返回了 lm.log 中的信息。

模型总体的 p 值十分小，修正 R 方达到 0.862 7，模型总体达到显著程度，是可信的。观察 Coefficients，模型的 5 个自变量和常数项的 p 值都小于 0.05，模型的残差值也处于 -1~1，因此整个模型是可靠的，不需要作出任何修改。

根据 Coefficients 提供的系数值，能够写出每个分类对应的回归方程。当花朵的类型为 setosa 时，有 $SL.sc = -0.5327 + 0.2610 \times SW.sc + 1.7678 \times PL.sc - 0.2901 \times PW.sc$ ；当花朵的类型为 versicolor 时，有 $SL.sc = -0.5327 + 0.2610 \times SW.sc + 1.7678 \times PL.sc - 0.2901 \times PW.sc + 1.2360$ ，将两个常数项相加后，有 $SL.sc = 0.7033 + 0.2610 \times SW.sc + 1.7678 \times PL.sc - 0.2901 \times PW.sc$ ；类似地，当花朵的类型为 virginica 时，有 $SL.sc = -0.1705 + 0.2610 \times SW.sc + 1.7678 \times PL.sc - 0.2901 \times PW.sc$ 。此时当 PL.sc 与 PW.sc 不变时，3 个类别对 SL.sc 的影响不再具有倍数关系。

在上述例子中，由 Species 拆出的两个哑变量的 p 值都通过了检验，因此可以很顺利地将两个哑变量均纳入线性回归模型中，但当分类变量的类型较多时，有时会出现一些哑变量能通过检验，而另一些哑变量不能通过检验的情况。这说明不能通过检验的那些哑变量的系数相似，回归模型分辨不出它们的区别。此时只需将不能通过检验的哑变量删除，只保留能通过检验的哑变量即可。

第 7 章 更高级的数据可视化

本章与第 3 章主题相同，都围绕数据可视化展开。但本章的内容专业性更强，是对第 3 章的延伸与拓展。与第 3 章相似，本章将介绍许多更高级的图形，并讨论它们的特性和它们所能解决的问题。通过阅读本章，读者也将进一步接触到 R 中强大的可视化系统。

7.1 基础图形的拓展与延伸

本节将讨论如何绘制更复杂的散点图、密度分布图和条形图。这些图形提供了比基础图形更丰富的功能，绘制起来也更加复杂一些。通过绘制这些图形可以更加深入地探索原始数据。

7.1.1 绘制分类散点图并添加图标

在有关相关分析和回归分析的章节中，我们已经发现分类变量和数值变量有很大的不同，处理时也要分开处理。在绘制图形时，通常我们也希望能够将不同类别的数据分开研究。比如，在绘制 iris 数据集的散点图时，将不同种类花朵的观测值分开绘制将取得更好的效果。

```
> attach(iris)
> Spe <- c("setosa","versicolor","virginica")
> Spe
[1] "setosa"      "versicolor"  "virginica"
> Spe<-as.factor(Spe)
> Spe
[1] setosa      versicolor  virginica
Levels: setosa versicolor virginica
```

上述代码首先绑定了 iris 数据集，然后创建了一个包含三个元素的向量 Spe。Spe 中存放了三种花朵类别，向量形式的 Spe 中每个元素都带有双引号，这表示它们是字符。这种形式与 iris 数据集中的元素形式并不完全一致，因此还需使用 as.factor() 函数将 Spe 转换为因子形式。再次查看 Spe 中的元素，此时每个元素中的双引号都已经去掉。

```
> col <- c("red","blue","orange")
> SW <- Sepal.Width[which(Species==Spe[1])]
> SL <- Sepal.Length[which(Species==Spe[1])]
> plot(SW,SL,col=col[1],xlim=c(2,4.5),ylim=c(4,8))
```

上述代码中第1行代码创建了一个向量，其中存储了三个颜色元素，将它们作为 iris 中三个类别的观测值的颜色。第2行和第3行代码分别将 Sepal.Width 和 Sepal.Length 中类别为 setosa 的观测值赋给了 SW 和 SL，其中 which() 函数实现了筛选功能，筛选条件中使用了双等号来表明筛选 Species 中与 Spe[1] 中元素相同的行，倘若在上一步中未把变量 Spe 转换为因子型，那么 which() 函数将判定 Species 中没有与 Spe[1] 相同的值，这一步中 SW、SL 将会是两个空向量。

最后一行利用 plot() 函数画出了 SW 和 SL 的散点图，并指定它的颜色为 col[1]，也就是红色，R 的返回结果如图 7.1 所示。参数 xlim 和 ylim 分别指定了散点图的 x 轴范围和 y 轴范围，这一范围是根据变量 Sepal.Width 和 Sepal.Length 的极值指定的，range() 函数可以查看向量的极值。

图 7.1 中的观测值呈带状分布在右下方，左上方留有一块空白，在这里添加新的点。倘若不手动指定坐标轴范围，R 会根据所有类别为 setosa 的观测值绘制图形，那么在添加类别为 versicolor 和 virginica 的观测值时就会超出范围。

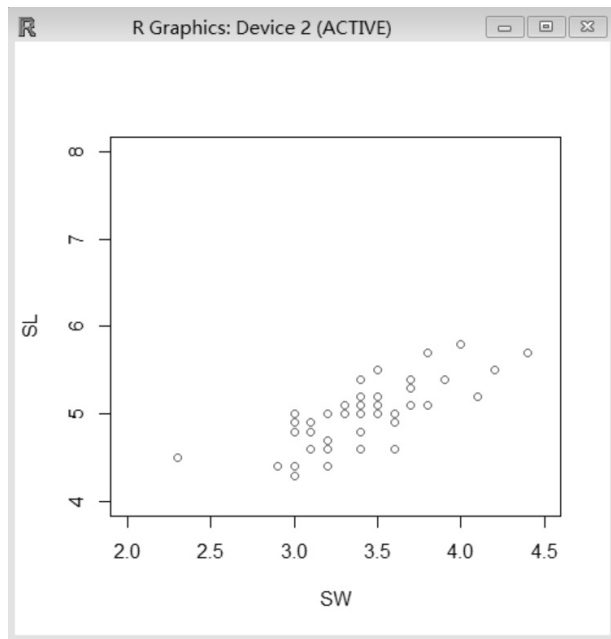


图 7.1 类别为 setosa 的观测值散点图

```
> for (i in 2:3){
+ SW <- Sepal.Width[which(Species==Spe[i])]
+ SL <- Sepal.Length[which(Species==Spe[i])]
+ points(SW,SL,col=col[i])}
> legend(3.8,7.5,legend=Spe,pch=1,col=col,bty="n")
```

points() 函数用于向图中添加点。上述代码中的 for 循环中的循环条件为 “i in 2:3”，即在 i 为 2 时循环一次，在 i 为 3 时循环一次。循环体中前两条语句筛选了 iris 中变量 Species 与 Spe[i] 相同的值的 Sepal.Width 和 Sepal.Length，并将值赋给了 SW 和 SL，

`point()` 函数则将 SW、SL 的散点以 `col[i]` 颜色添加到散点图上。`legend()` 函数则为整张散点图添加图标。

图 7.2 是 R 的返回结果。与图 7.1 对比，此时图形中已增加了更多的点，并添加了一个图标。在上述代码中，`for` 循环一共执行了两次，分别在图中添加了 `versicolor` 类别的蓝色散点和 `virginica` 类别的橘色散点。

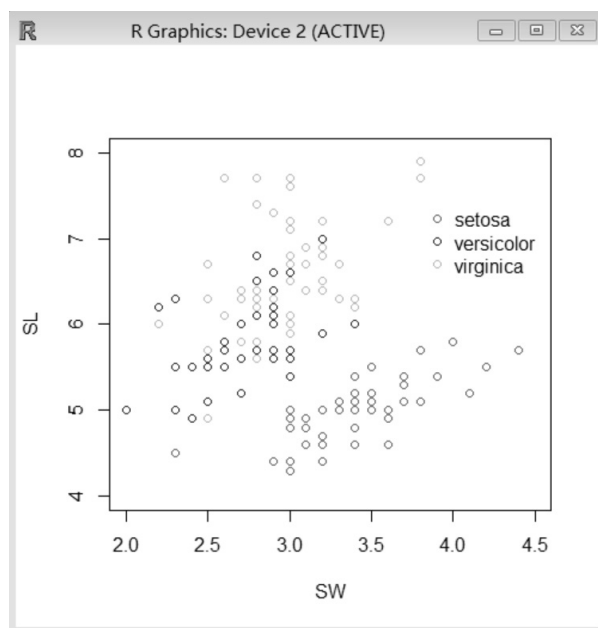


图 7.2 3 种类型观测值的散点图

`legend()` 函数的前两个参数指定在坐标 (3.8, 7.5) 处添加图标，由于之前已经创建了存储类别信息和颜色信息的 `Spe` 向量和 `col` 向量，因此在 `legend()` 函数中只需直接调用 `Spe` 向量和 `col` 向量，将它们作为图标的名称和颜色即可。参数 `pch` 指定图标符号的类型为第一类，参数 `bty` 则指定了图框的类型为无。

观察图 7.2，`setosa` 类别的点集中在图形的右下方，`versicolor` 集中在左下方，`virginica` 则集中在右上方。此外，`setosa` 类别与另外两种类别的点具有较为明显的分界线，而 `versicolor` 和 `virginica` 则有部分数据混杂在一起。图 7.2 是根据 `Sepal.Width` 和 `Sepal.Length` 变量绘出的散点图，`iris` 还提供了另外两个变量，将 4 个变量结合起来能绘出更多的散点图。

7.1.2 绘制含多种类别的密度分布图

密度分布图的作用非常强大，在第 3 章中已经提过，直方图和密度分布图都能反映一份数据的分布状况，而数据的分布又决定了该数据适合怎样的数据分析方法。对分类数据来说，研究不同类别的分布是否具有差异是较有意义的。均值 `t` 检验和方差分析都能检验两个或多个总体是否服从同一分布，但程序包 `ggplot2` 提供了一个分类绘图的函数，能更方便、直观地同时反映几个总体的分布情况。


```
> library(ggplot2)
> ggplot(iris,aes(x=Sepal.Length,fill=Species))+geom_density()
```

`ggplot()` 函数的格式十分简洁。上述代码首先载入了 `ggplot2` 包，然后将 `ggplot()` 函数和 `geom_density()` 函数组合起来绘制了如图 7.3 所示的密度分布图。`ggplot()` 函数指定了两个参数，第一个参数指定 `ggplot()` 函数的应用对象是 `iris` 数据集，第二个参数 `aes` 指定图形的坐标轴参数，参数 `aes` 内嵌了两个函数，参数 `x` 指定图形以 `Sepal.Length` 为 x 轴，参数 `fill` 则指定按照 `Species` 分类。显然，`fill` 是一个非常重要的参数，如果不加这个参数，`ggplot()` 将绘出一个总的密度分布图。

指定好 `ggplot()` 函数后，上述代码使用加号在代码后方加上了 `geom_density()` 函数。与书写 `for` 循环时不同，此处的加号需要分析人员自己输入，它代表在同一个图形中应用这两个函数。`ggplot()` 函数仅给出了图形的基本设置，`geom_density()` 函数则进一步制定绘图类型为密度图。这两个函数不能拆成两句命令逐一输入，否则 R 将报错。

观察图 7.3，属于不同类别观测值的 `Sepal.Length` 显然具有不同的密度分布，图 7.3 中用三种不同的颜色清楚地将三个分布标识了出来。`setosa` 的红色分布有最小的密度均值，它的峰值也是最高的，显然，它的数据更集中一些。`versicolor` 和 `virginica` 的峰值高度较为相似，但 `versicolor` 的密度均值较 `virginica` 更小，即 `versicolor` 中的数据普遍更小。

```
> ggplot(iris,aes(x=Sepal.Length,fill=Species))+geom_density()+facet_grid(Species~.)
```

图 7.3 画出的图形尽管已经将 `Sepal.Length` 按照类别画出了三个不同的密度分布图，但 `setosa` 和 `versicolor` 的密度分布分别被挡住了一部分，`facet_grid()` 函数能将不同的图形分别画在不同的图形中。

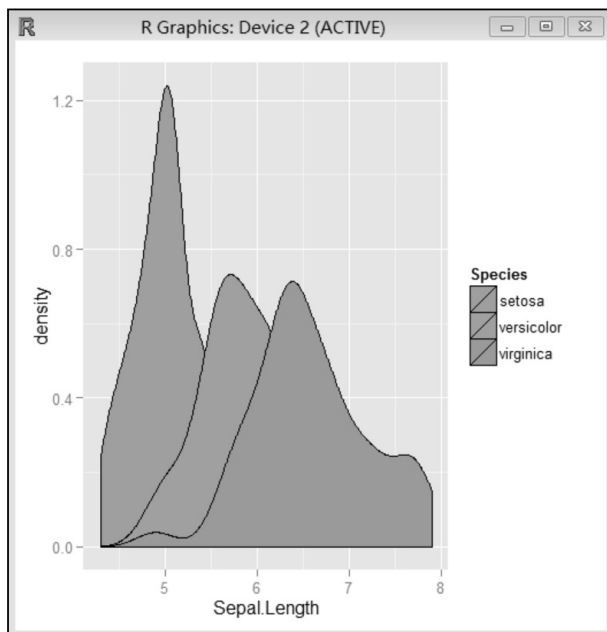


图 7.3 三种类别花朵的密度分布图

上述代码在之前的代码上再次添加了 `facet_grid()` 函数，它仅有一个参数 `Species~.`，符号 `~` 左侧的 `Species` 表示图形将 `Species` 的类别看作列，符号 `~` 右侧的 `.` 表示忽略按行分类的变量，当此处也填上分类变量的名称时，`ggplot()` 函数将绘出一个矩阵，比如 `facet_grid(Species~Species)` 会绘出一个 3 行 3 列的大矩阵，大矩阵中包含 9 个图形，每个图形都是对 `Species` 按类别交叉分组后取子集绘出的图形。

图 7.4 是最终的绘图结果，`Sepal.Length` 按照三种类型分别绘出了三幅密度分布图。观察图 7.4，`setosa` 具有最集中的数据分布，它的数据基本全部集中在 4~6 之间，`versicolor` 和 `virginica` 的数据分布较为相似。这三个密度分布图都接近钟形，与正态分布较为相似。此外，`setosa` 的左侧和 `virginica` 的右侧有一个“悬崖”般突然的截断，这说明 `iris` 中的数据很可能抛掉了太小和太大的数据。

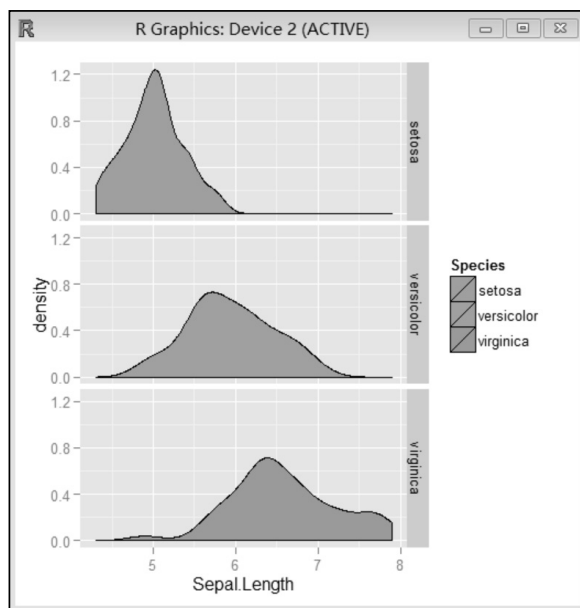


图 7.4 分别绘制三种类别花朵的密度分布图

7.1.3 复合条形图和堆栈条形图

条形图是另一类重要的基础图形，对条形图的拓展和延伸主要有复合条形图和堆栈条形图。这两种图形的绘制都较为容易，较难的部分在于如何将数据转换为合适的类型。

```
> attach(Orange)
> head(Orange)
  Tree age circumference
1    1  118             30
2    1  484             58
3    1  664             87
4    1 1004            115
5    1 1231            120
```

```
6      1 1372      142
> barplot(circumference,col="white")
```

Orange 数据集中包含三个变量，其中 `Tree` 和 `age` 是分类变量，分别存放了橘子树的编号和年龄，`circumference` 则是数值型变量，存放了橘子树的树围。上述代码首先绑定了 Orange 数据集，然后查看了 Orange 数据集中的前 6 行数据，显然，橘子树的树围观测值已经按照树的编号和年龄规规矩矩地存放在一个矩阵中。`barplot()` 函数绘制了 `circumference` 的复合条形图，并指定它的颜色为白色。

图 7.5 是上述代码的返回结果，很明显，图 7.5 中的条形以 7 个长条为一组，总共分为 5 组。每一组内的 7 个长条都逐一递增，一条比一条长。从实际意义来看，这 5 组数据对应着 5 棵橘子树，每组数据中的 7 个长条则对应取得观测值的 7 个年份，显然，橘子树的树围只会随着年份的增加而增加，而不会随着年份的增加而减少。

此外，这 5 棵橘子树树围的最小值较为相似，而最大值则具有较大的差异，其中，第 3 组数据的最大值和第一组数据的最大值比较接近，是 5 个最大值中最小的两个，第 2 组和第 4 组则具有最大的最大值。这 5 组数据中，前 4 年的数据增长幅度和第五年到第六年的数据增长幅度都较大，第 4~5 年，以及第 6~7 年的增幅则较小，这和树木前期树围增长较快有关，也和观测年份间隔的大小有关。

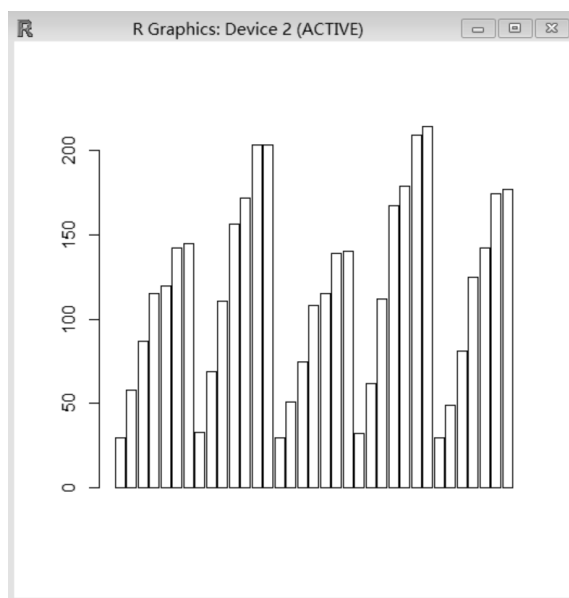


图 7.5 `circumference` 的复合条形图

复合条形图的绘制较为容易，而堆栈条形图则要求将数据按照观测值一一放到矩阵中。

```
> library(reshape)
> bar <- cast(Orange,age~Tree,value="circumference")
> bar
   age    3    1    5    2    4
```

```

1  118  30  30  30  33  32
2  484  51  58  49  69  62
3  664  75  87  81 111 112
4 1004 108 115 125 156 167
5 1231 115 120 142 172 179
6 1372 139 142 174 203 209
7 1582 140 145 177 203 214

```

上述代码首先载入了 `reshape` 包，然后用 `cast()` 函数将 `Orange` 转换为以 `age` 为列、以 `Tree` 为行、以 `circumference` 为值的数据框 `bar`，最后一行命令查看了 `bar` 中的数据。此时 `bar` 中共有 6 列变量，第 1 列变量记录了每一行数据对应的年份，这一列数据是我们所不需要的。此外，`barplot()` 函数只接受向量或矩阵类型的输入变量，因此还需将 `bar` 转换为合适的格式。

```

> bar <- bar[-1]
> bar <- as.matrix(bar)
> colnames(bar) <- c(3,1,5,2,4)
> bar
      3  1  5  2  4
[1,] 30 30 30 33 32
[2,] 51 58 49 69 62
[3,] 75 87 81 111 112
[4,] 108 115 125 156 167
[5,] 115 120 142 172 179
[6,] 139 142 174 203 209
[7,] 140 145 177 203 214

```

上述代码中的第 1 行命令去掉了 `bar` 中的第 1 列，第 2 行命令则使用 `as.matrix()` 函数将数据框 `bar` 转换为矩阵形式。由于矩阵转换会丢失 `bar` 的所有名称，因此第 3 行命令重新为 `bar` 命名，将 `bar` 的 1~5 列分别按照每一列对应的橘子树编号命名。再次查看 `bar` 中的数据，此时 `bar` 已经转换为一个具有合适名字的矩阵。

```

> for(i in 6:1){
+ bar[i+1,] <- bar[i+1,]-bar[i,]}
> bar
      3  1  5  2  4
[1,] 30 30 30 33 32
[2,] 21 28 19 36 30
[3,] 24 29 32 42 50
[4,] 33 28 44 45 55
[5,]  7  5 17 16 12
[6,] 24 22 32 31 30
[7,]  1  3  3  0  5

```

对数据的最后一步转换是将每年的树围数据转换为每年树围的增长数据。上述代码中使用 `for` 循环实现了这个功能。在上述代码中， i 从 6 循环到 1，每次循环都将 $i+1$ 行的数据赋为了第 $i+1$ 行与第 i 行数据的差。这种循环一共执行了 6 次，倘若令 i 从 1 循

环到 6, `bar[i+1,]` 减去的将是修改后的 `bar[i,]`, 将得不到正确结果。

```
> barplot(bar,col=c("blue","steelblue","skyblue","orange","yellow","grey",
"white"))
```

准备好原始数据有后, 使用 `barplot()` 函数绘制堆栈条形图是一件非常容易的事。上述代码使用 `col` 参数指定了 7 种颜色赋给堆栈条形图, 如果不设定参数 `col`, `barplot()` 函数将使用由深到浅的灰色来区分每一个条形的不同部分。

观察图 7.6, 条形的前 4 块分块和第 6 块分块都较大, 在这些年份中橘子树的树围增长也比较大, 而第 5 块分块和第 7 块分块则小得可怜, 这些年份中橘子树的树围也确实几乎没有增长。这一结论与图 7.5 的结论十分吻合, 也反映了 `circumference` 在不同观测年份中的增长情况。

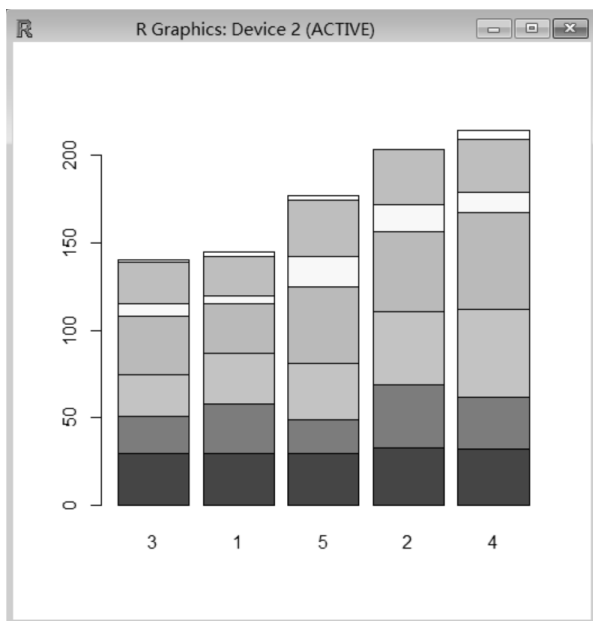


图 7.6 `circumference` 的堆栈条形图

7.2 有关多元分布函数的特殊图形

本节的主题是多元分布函数的体现。在有关基础图形的章节中不难发现基础图形的最大的局限就是它们不能表现多于二维的数据。本节将讨论如今最流行的 4 种表现多元分布的特殊图形, 以弥补基础图形的不足。

7.2.1 星图和脸谱图

对于维数多于三维、不超过 6 维的数据集来说, 星图是反映数据分布的最佳图形。R 的基础包中提供了绘制星图的函数, 只需调用即可绘出星图。

```
> attach(iris)
> stars(iris)
```

仍以 iris 数据集为例，iris 数据集中一共有 5 列变量：Sepal.Length、Sepal.Width、Petal.Length、Petal.Width、Species，即 5 个维度。stars() 函数用于绘制星图，图 7.7 是上述代码的返回结果。观察图 7.7，它由 150 个元素组成，每一个元素都是一个小星星，对应 iris 中的一个观测值。观察这 150 个星星，它们中的每一个都由 5 个角组成，每个角都由一条轴线和中心点连接了起来，显然，5 条轴线分别对应 5 个维度的值，值越大，轴线就越长，画出的星星也就越大。

图 7.7 中的星星明显分为小、中、大三类，前 50 个星星最小，中间的 50 个星星较大一些，后 50 个星星最大。根据星星的形状可以将 iris 中的观测值分为三组。这三组观测值恰好对应 Species 的三种分类。

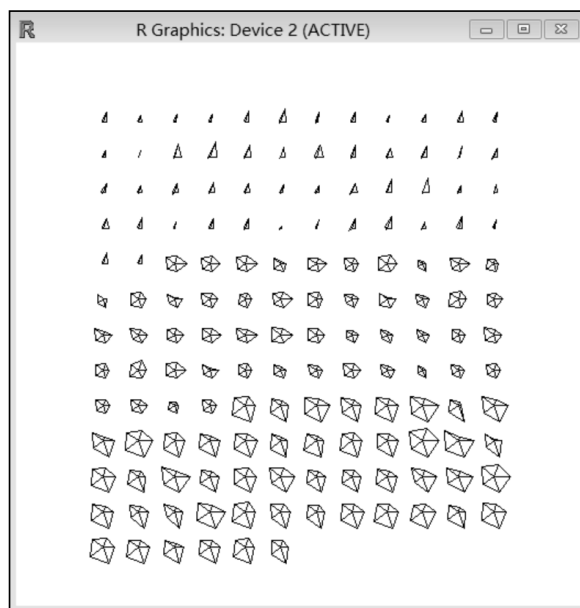


图 7.7 使用 iris 数据集绘制星图

```
> stars(iris[1:20,-5],full=FALSE,draw.segments=TURE)
```

上述代码在 stars() 函数中添加了更多的参数。在图 7.7 中将 iris 中的 Species 变量也绘制了出来，这是一个分类变量，为它绘制星图的意义并不大，因此上述代码中第一个参数指定为 iris[1:20,-5]，即绘图时去掉 iris 中的第 5 列变量，并只绘制 iris 的前 20 个观测值。参数 full 指定为假，此时绘制的将不再是一整个星星，而是半个星星，这是由于我们现在仅绘制 4 个变量，因此半个星星的范围即足够，画为一个星星反而过于分散，不利于观察。第三个参数指定参数 draw.segments 为真，则使用弧线连接轴线。

图 7.8 是上述代码的返回结果，与图 7.7 相比，图 7.8 中的星星个数较少，R 自动为每个星星加上了序号。现在图 7.8 绘出的是一个一个小扇形构成的类似半圆的元素，它与饼图有些相似，但饼图是用每个扇形的面积来反映观测值各个维度的大小的，而星图

则使用每个扇形的半径来反映。参数 `draw.segments` 还自动为每个扇形涂上了颜色，这在维度较少时有利于观测，但在维度较多时则会起到反作用。

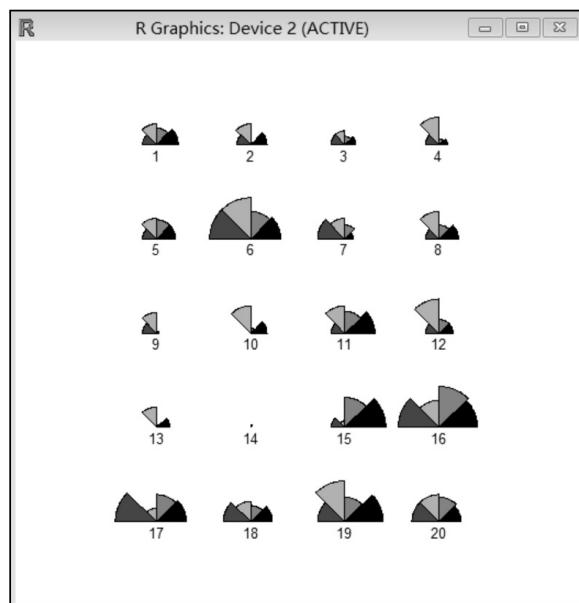


图 7.8 绘制新的星图

脸谱图是与星图类似的一种图形，它创建的图形与星图相似，但它使用了比星图更多的维度。

```
> library(aplpack)
> faces(iris[1:30,-5])
effect of variables:
modified item      Var
"height of face"   "Sepal.Length"
"width of face"    "Sepal.Width"
"structure of face" "Petal.Length"
"height of mouth"  "Petal.Width"
"width of mouth"   "Sepal.Length"
"smiling"          "Sepal.Width"
"height of eyes"   "Petal.Length"
"width of eyes"    "Petal.Width"
"height of hair"   "Sepal.Length"
"width of hair"    "Sepal.Width"
"style of hair"    "Petal.Length"
"height of nose"   "Petal.Width"
"width of nose"    "Sepal.Length"
"width of ear"     "Sepal.Width"
"height of ear"    "Petal.Length"
```

程序包 `aplpack` 提供了专门用于绘制脸谱图的 `faces()` 函数，上述代码首先加载了 `aplpack` 包，然后调用 `faces()` 函数绘出了脸谱图，与 `stars()` 函数相似，`faces()` 函数使用

参数 `iris[1:30,-5]` 指定使用除第 5 列变量外的其他变量为 `iris` 中前 30 个观测值绘制出脸谱图。R 同时返回了一个列表和一张脸谱图，其返回结果如图 7.9 所示。

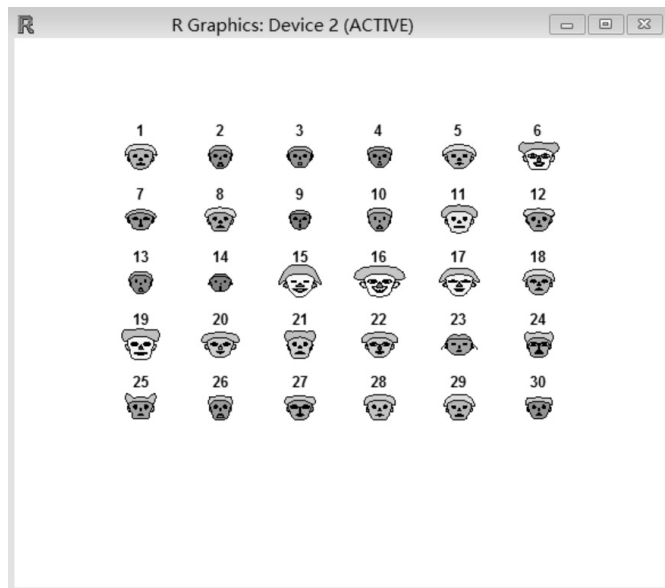


图 7.9 使用 `iris` 绘制脸谱图

R 返回的列表显示绘制脸谱图一共用了 12 种统计量，分别是脸的高度、宽度和结构；嘴唇的厚度、宽度和上翘角度；眼睛的高度、宽度；头发的高度、宽度和样式；鼻子的高度、宽度；耳朵的高度、宽度，返回列表中同样也给出了每一种数据是根据 `iris` 中的哪种变量绘制的，显然，`iris` 中的 4 个数值型变量被轮流使用了三次。利用这 12 组数据，R 绘出了图 7.9。

图 7.9 绘出了 30 张脸谱，这些脸谱的脸型、五官、发型、颜色都有较大的差异，显然，脸谱越相似，就说明对应的观测值越相似。`faces()` 函数还提供了用于控制行数和列数的参数，它也能画出其他类型的脸谱。与星图相比，`faces()` 函数显然更适合那些变量维数超过 6 维的数据，当数据多于 12 维时，它同样能在脸谱上添加新的特征用于反映数据结构。

7.2.2 轮廓图

轮廓图是一类用于表现多维变量的漂亮图形。程序包 `lattice` 提供了与之相关的函数。

```
> library(lattice)
> parallelplot(iris[1:4])
```

上述代码首先载入了 `lattice` 包，然后使用 `parallelplot()` 函数绘制了 `iris` 中前 4 列变量的轮廓图。返回结果如图 7.10 所示。观察图 7.10，其横轴简略地给出了数值的最大值和最小值，纵轴则标出 4 条水平线，分别代表 `Sepal.Length`、`Sepal.Width`、`Petal.Length` 和 `Petal.Width` 4 个变量。一些彩色的折线穿过了这 4 个变量，每个折线都有两个折点和两

个端点。实际上，一条彩线就对应了一条观测值，而折点和端点则对应了观测值在 4 个变量上的取值。

为了区分不同观测值，轮廓图使用多种色彩标注出不同的线条。观察图 7.10 中的彩色线条，Sepal.Length 和 Sepal.Width 的值分散得较为均匀，看不出什么规律，但 Petal.Length 和 Petal.Width 的值则分别集中在两个极值附近，较少的一部分数据集中在极小值附近；较多的一部分数据集中在极大值附近。

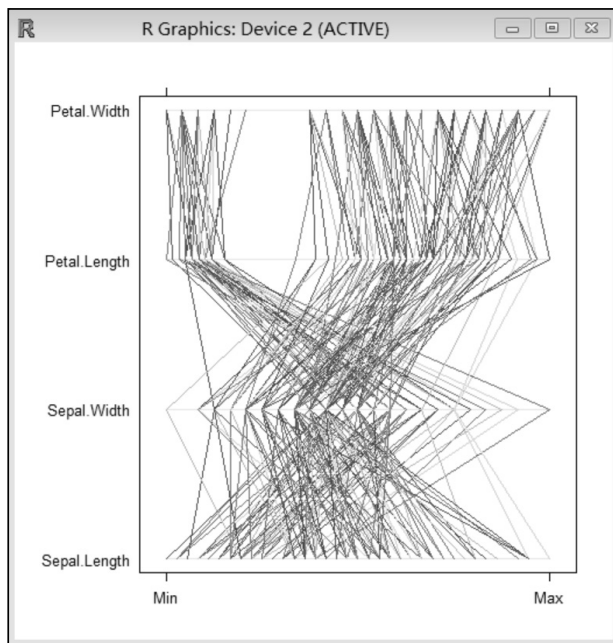


图 7.10 使用 iris 绘制轮廓图

```
> par(las=1)
> parallelplot(iris[1:4], groups=Species, horizontal.axis=FALSE)
```

上述代码为 `parallelplot()` 函数添加了更多的参数。其中第 1 行代码使用 `par()` 函数指定坐标轴的坐标名称横向排列，否则图 7.11 中的横轴坐标名称将会垂直排列，可读性将变差。在第 2 行代码的 `parallelplot()` 函数中共指定了三个参数，第一个参数同样指定绘图对象为 `iris` 的前 4 列变量，第二个参数 `groups` 则指定按照 `Species` 变量分组绘图。第三个参数指定 `horizontal.axis` 为假，即绘制一个横放的图形，将变量名放在横轴上。

观察图 7.11，图中使用三种颜色绘出了分属三种类别的观测值数据。同种颜色的线条聚集在一起，不同颜色的线条则彼此分离，说明不同种类的观测值数据有较大的差异。此外，蓝色线条的 `Sepal.Length` 值聚集在最下方，粉色线条的 `Sepal.Length` 值聚集在中间，绿色线条的 `Sepal.Length` 值聚集在最上方，联系图 7.3 和图 7.5，蓝色线条、粉色线条和绿色线条分别对应 `setosa`、`versicolor` 和 `virginica`。从变量角度来考虑，轮廓图同样也反映出变量的分布情况，显然，`Petal.Length`、`Petal.Width` 对于分辨不同观测值所属的种类有帮助，而 `Sepal.Length`、`Sepal.Width` 的作用则较小。

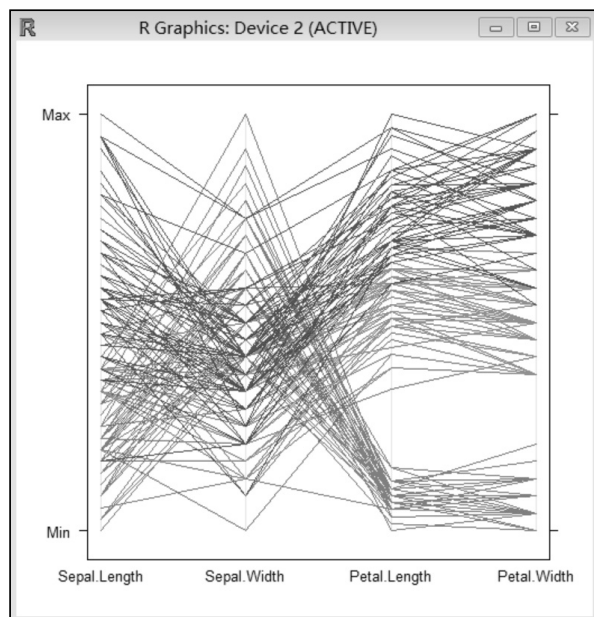


图 7.11 绘制新的轮廓图

与星图和脸谱图相比，轮廓图同样保留了数据集中的全体信息，但它并不随着数据的增多而显得如图 7.7 一样臃肿、让人抓不住重点。显然，星图和脸谱图适合观测值较少的情况，而轮廓图更适合观测值较多的情况；星图和脸谱图在观测值没有分类或分类情况未知时适用，而轮廓图则在观测值分为几个大类时更适用。上述规则的唯一特殊情况是在观测值非常少，且每个都自成一类时（比如观察 10 个学生的 5 门成绩），星图和脸谱图没什么用，轮廓图则表现突出。

在绘制轮廓图时最后需要注意的一点是变量的排列顺序。不同的顺序会达到不同的效果。以图 7.11 为例，蓝色线条的 `Spetal.Length` 值较高，其他三个变量的值都较低；粉色线条和绿色线条的 `Spetal.Length` 值较低，其他三个变量的值都较高，因此将 `Sepal.Width` 与 `Spetal.Length` 的顺序调换后绘出的图将更合理。绘制轮廓图的主要规则是将相似的变量尽量排到一起，这一点在变量个数达到十几个时更为重要。

7.2.3 调和曲线图

在介绍轮廓图时，当变量个数增多时，轮廓图将不能很好地归纳出变量中的信息，为了弥补轮廓图的这一缺陷，安德鲁斯在 1972 年提出了能将多维变量映射到二维平面上的调和曲线公式。

不妨设每个观测值都有 p 维变量，分别用 (x_1, x_2, \dots, x_p) 表示，其中 x_i 代表观测值在第 i 个变量处的值。那么调和曲线公式可以写为 $f(t) = \frac{x_1}{\sqrt{2}} + x_2 \cdot \sin(t) + x_3 \cdot \cos(t) + x_3 \cdot \sin(2t) + x_4 \cdot \cos(2t) + \dots$ ，公式中的 t 可以取位于区间 $[-\pi, \pi]$ 中的任意值。以 t 为横轴， $f(t)$ 为纵轴，每个观测值都能绘制出一条调和曲线，将所有的曲线绘制在一张图上，即可得到调和曲线图。

```
> library(MSG)
> andrews_curve(iris[, -5], col = as.integer(iris[, 5]))
```

程序包 MSG 提供了 `andrews_curve()` 函数用于绘制调和曲线图，上述代码加载了 MSG 包，并调用 `andrews_curve()` 函数绘制了如图 7.12 所示的调和曲线图。

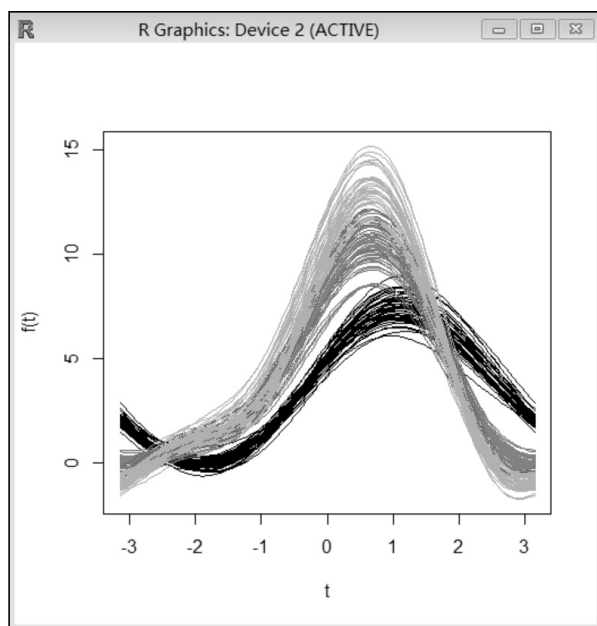


图 7.12 使用 iris 绘制调和曲线图

在上述代码中 `andrews_curve()` 函数一共指定了两个参数，第一个参数指定绘图对象为 `iris` 的前 4 列变量，第二个参数 `col` 指定按照 `iris[, 5]` 的分类为曲线绘制颜色，这一参数的作用类似于轮廓图中的 `groups` 参数，如果去掉这一参数，图 7.12 中的曲线颜色将变得杂乱无章。`andrews_curve()` 函数中其他常用的参数还有曲线中折点的个数，折点越多，曲线就越光滑。

观察图 7.12，图中的曲线被分为绿色、红色和黑色三种颜色，每种颜色的曲线都紧紧地拧成一股。绿色曲线和红色曲线挨得较近，黑色曲线则与其他曲线分离得更明显。观察计算调和曲线的公式，它能计算无穷多个变量，并将无穷多个变量转化到二维平面上。因此调和曲线是唯一一种不用担心观测值的变量过多而影响整洁美观性的图形。

但调和曲线也忽略了一些信息，它只能对比观测值之间的相似程度，而不能对比变量之间的相似程度，也表现不出哪些变量有助于为观测值分类，哪些变量反而混淆了观测值类别。总的来说，调和曲线图在聚类分析中有很重要的作用，无论观测值的类别已知还是未知，调和曲线图都能比轮廓图更好地反映出观测值的相似程度。

7.3 建立最简单的 3D 图形

常用图形中最后一类特殊的图形是三维图形。三维空间是人类所能直观感受到的最

高维空间，超过三维以后的数据将不再具有几何意义。绘制 3D 图形要比绘制 2D 图形复杂得多，但 3D 图形的信息量也比 2D 图形丰富许多，R 提供了一些函数专门用于绘制 3D 图形。

3D 散点图是最简单的 3D 图形，R 中的基础包中并未提供有关 3D 制图的函数，程序包 `scatterplot3d` 提供了用于绘制 3D 散点图的函数。

```
> attach(iris)
> library(scatterplot3d)
> scatterplot3d(Sepal.Length, Sepal.Width, Petal.Length)
```

上述代码首先绑定了 `iris` 数据集并加载了 `scatterplot3d` 程序包，然后使用 `scatterplot3d()` 函数绘制了一张 3D 散点图，`scatterplot3d()` 函数中的三个参数按顺序指定了 x 轴、 y 轴、 z 轴使用的数据。图 7.13 是 R 的返回结果，它绘制了一个立方体，图标给出了三个坐标轴的刻度，同时也指出 x 轴、 y 轴和 z 轴分别对应 `Sepal.Length`、`Sepal.Width` 和 `Petal.Length`。

在图 7.13 中的三维空间中呈带状分散着 150 个散点。这些散点中的三分之一落在左下方的位置，另外的三分之二落在右上方的位置。为了增强立体感，3D 散点图还在图形的下方增加了表格，用于帮助确定每个散点对应的位置。`scatterplot3d()` 函数还提供了用于修改坐标轴的颜色、刻度、名称、大小等一系列参数，以及用于设置主标题、子标题的参数等。

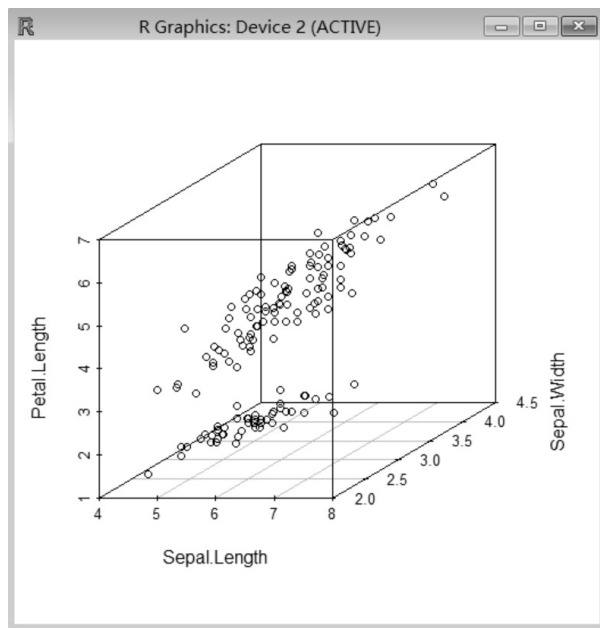


图 7.13 使用 `iris` 绘制 3D 散点图

除散点图外，曲面图也是一种非常适合用三维空间表现的图形。R 中的 `lattice` 程序包提供了 `wireframe()` 函数用于绘制曲面图，曲面图不仅能表现地理数据，对其他的一些分类数据同样具有很强的表现力。

```
> attach(Orange)
> library(lattice)
> wireframe(circumference~Tree*age, zlab="circum")
```

上述代码首先绑定了 Orange 数据集并加载了 lattice 程序包，然后用 wireframe() 函数绘制了如图 7.14 所示的曲面图，wireframe() 函数中的第一个参数指定 circumference 为 z 轴，Tree 和 age 为 x 轴和 y 轴，第二个参数则指定 z 轴的名称为 circum，这种缩写的方式避免了 z 轴名称过长，超出图形框的范围。

图 7.14 所示的图形好似一张被折过的纸，每一个折点都代表一个值。观察 y 轴，随着 age 的增长，circumference 的值在一路攀升；观察 z 轴，每条折线则表现了不同树木在同一年份的树围长度。通过图 7.14，circumference 和 Tree、age 的关系同时表现了出来，Tree 和 age 之间是否有交互作用也能从图 7.14 中观察得到。

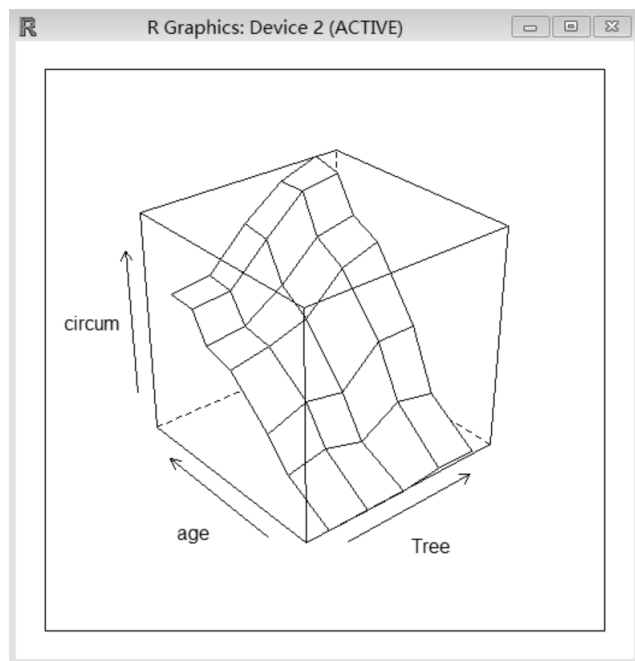


图 7.14 使用 Orange 绘制曲面图

在曲面图中 z 轴是被着重表现的坐标轴，它所对应的变量往往是我们想要观察的主要变量。与 3D 散点图不同，随意调换曲面图中坐标轴对应的变量很可能会得到奇怪的结果，而 3D 散点图则不存在类似的问题。wireframe() 函数同样提供了设置曲面颜色的参数及其他有关坐标轴的常规参数。

7.4 如何让图形更美观

对希望在数据可视化方面有所建树的读者来说，本节的内容格外重要。对数据分析师来说，从原始数据中探索出结论十分重要，用合适的数据可视化将分析结论表现出来

更为重要。好的数据可视化能够起到画龙点睛的作用，将一份晦涩的结论表现得活灵活现。

对于一份数据来说，选择合适的图形是最重要的一步。气泡图可以在相同的空间中表现更多的数据，饼图能够突出局部和整体的关系，而条形图在简洁明了方面具有特殊的优势，其他的基础图形也各有各的特点。

条形图有丰富的分类，也很容易和其他图形组合起来，因此条形图一直是最受欢迎的图形。图 7.15 是一张条形图和线图的组合图形，图中的条形图是复合条形图，其中蓝色的长条表示汽车销量，红色的长条表示乘用车销量，绿色的线条则表示商用车销量，而绘在条形图上的线图则为汽车同比增长率。

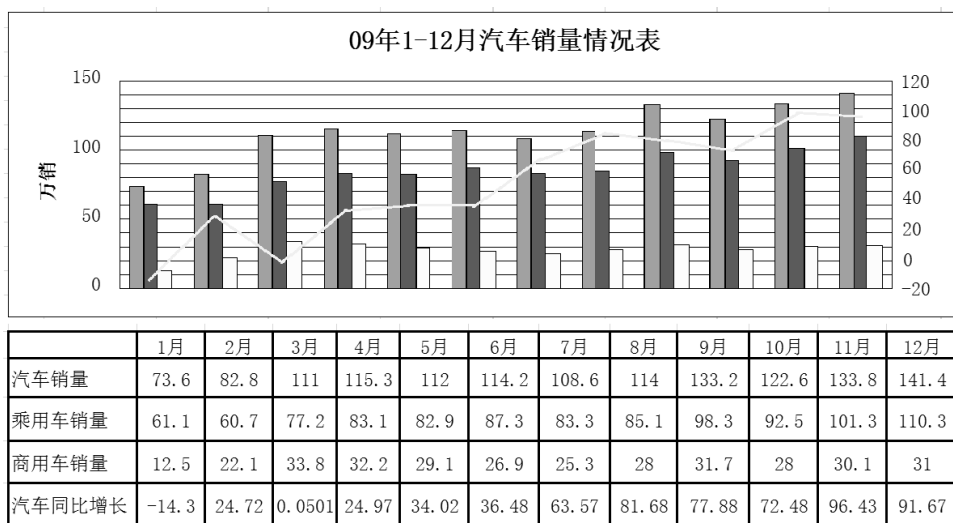


图 7.15 条形图和线图的组合图

图 7.15 的下方给出了有关汽车销售的详细数据，同时组合图形也绘制了三条坐标轴用以标出月份、汽车销量和增长率。这种组合图在一张图形中提供了更多的信息，此外在数据趋势的对比上也提供了方便。线图和条形图是一组较好的搭档，线图和散点图搭配也能产生不错的效果，其他好的组合还有树状图搭配气泡图等。

绘制图形的第二条重要步骤是绘制合适的图标。图标是一个图形中必不可少的存在，比如图 7.15 就标出了复合条形图中每种颜色的长条的含义。宽泛地说，图标也包含坐标轴和标题，如果没有图标，读者就不能理解图形的具体含义。

以数据可视化专家的专业眼光来看，图 7.16 使用了过于华丽的图形，这可能会将读者的注意力从真正重要的东西上引走，而唯一能够避免这一结果的就是图 7.16 中的图标。与朴素的条形图相比，漂亮的图片能给人以视觉享受，但绝不能为了好看而抛弃数据可视化的主角，即数据本身。图 7.16 左侧的图标详细标注了图中环形的每种颜色所代表的国家，也为图中粗细不一、深浅不一的曲线逐一标出了来源，再加上图形左侧大幅的文字说明，图 7.16 才得以在这张过分华丽的图形中突出重点。

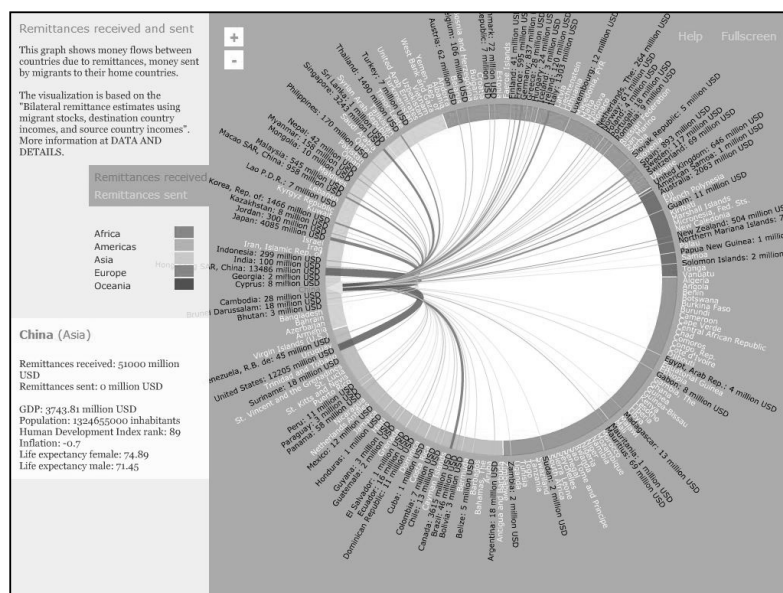


图 7.16 利用图标创建优秀图形

图 7.16 使用的另一个技巧是色彩的对比，图中使用黑色标出了 USD 类别的数据，使用白色标出了其他类别的数据。指向 China 的曲线也用颜色的深浅区分了重要程度。色彩是图形中一个较不起眼、但同样重要的元素。

图 7.17 是一幅关于民众对于国家经济的观点的图形。这张图片有两个维度，横轴代表民众对国家经济现状的评价，纵轴代表民众对国家能否好转的期望。图 7.17 由红色和蓝色构成，认为国家经济越差、好转越难的观点所对应的颜色越红；认为国家经济越好，好转越容易的观点所对应的颜色越蓝。图 7.17 通过颜色的渐变反映了观点的逐渐转变，在图 7.17 中也给出了持每种观点的民众所给出的高频词，同样有助于理解图形。这种热力图不仅能用于民众意见的调查，同样也能和地图相结合以用于标注热点事件。

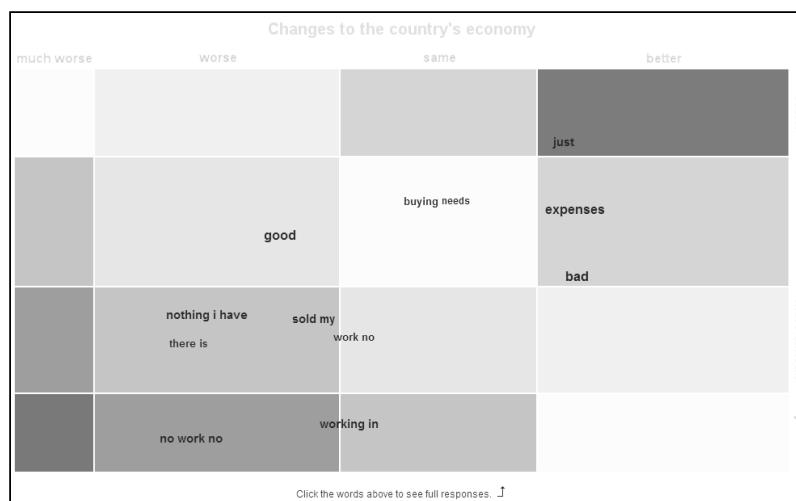


图 7.17 经济评价热力图

数据可视化是一门宏大的学科，它是一门注重细节的学科，本节所提到的内容不过是冰山一角，与数据可视化有关的重要准则还有许多，本节只列出了其中最重要的三条。R 提供的绘图系统能够满足人们日常的需求，对希望绘出更精美的图形的读者来说，笔者在此推荐 Geckboard、Mixpanel、Optimizely、Qualaroo 等软件。

7.5 更多的绘图包和系统

有关 R 的绘图系统前面已经讨论了许多程序包和函数，下面不妨专门拿出一节的篇幅来总结一下。R 中最著名的绘图系统共有三个，分别是 R 的基础绘图包 `graphics`、还有高级绘图包 `ggplot2` 和 `lattice`。程序包 `graphics` 提供的绘图函数较为简单，能满足大部分绘图需求，但较为粗糙。

绘图包 `ggplot2` 是 `ggplot` 包的升级版，其作者是著名的数据可视化专家。在上文中用到 `ggplot2` 中的函数，`ggplot2` 采用图层的思想来绘制图形，即将一张图看作一个一个的小图层，比如坐标轴和标题是一个图层，条形图是一个图层，线图又是一个图层。将许多图层叠加起来后就得到了完整的图形。`ggplot2` 包还认为图形的美化分为美化数据本身和美化与数据无关的东西，前者如修改直方图的宽度，后者如修改直方图的颜色。总的来说，`ggplot2` 的绘图思想十分专业，绘出的图形也兼具简洁美观和信息丰富两个优点。

绘图包 `lattice` 是一个通用的绘图包，它能十分方便地嵌入 MATLAB、C、C++ 等编程语言中。`lattice` 包在绘制多元数据上尤其擅长，它是一个基于网格的系统，这意味着 `lattice` 绘出的图形非常精致，它特别附加了多元绘图系统，强调高维数据的绘制。此外，`lattice` 绘出的图形风格简明，十分适合出版，许多专业论文都使用 `lattice` 绘制图形。

除上述三个绘图系统外，R 也提供了一些小而美的绘图程序包。与二维散点图有关的是 `car` 包的 `scatterplot.matrix()` 函数；利用 `cwhplot` 包的 `pltSplomT()` 函数能像 `pair()` 函数那样画出散点图矩阵，但利用它也能画出柱状图矩阵和密度估计图矩阵。与 3D 绘图有关的包是 `scatterplot3D` 和 `aplpack` 包，此外 `misc3d` 包有可视化密度的函数。针对多元数据可视化技术开发的包有 `YaleToolkit` 包和 `agsemisc` 包。

另外，一些特别的包有可画椭圆的 `ellipse` 包；为多元可视化提供水平集树形结构的 `denpro` 包；提供针对聚类的散点图和平行坐标图的 `gclus` 包；还有能绘出高质量图形的包等。最有趣的是 `iplots` 包提供的动态交互图，以及 `seriation` 包提供的能重新排列矩阵和系统树的 `seriation` 方法。

R 绘图系统的最后一个成员是一些统计分析包中特别配置的绘图函数。比如专门用于添加回归线的 `lm()` 函数，以及第 8 章马上要介绍的系统聚类树状图的绘制。这些函数提供了专业性非常强的特殊图形，解读起来较为困难，却有着不能替代的重要作用。

第 8 章 R 中的聚类分析和判别分析

本章的主题是聚类分析。聚类分析是我们接触的第三个偏重机器学习方向的统计方法，它主要应用于较大的数据集上。本章将介绍目前主流的聚类分析，并着重讲解其中较为简单的三种，即 KNN 聚类、系统聚类和快速聚类。还将讨论聚类分析和判别分析的异同。

8.1 几种聚类分析的异同

聚类分析是一种机器学习领域中最常用的分类方法，在聚类分析中，观测值的类别通常是未知的，我们希望将观测值聚为合适的几个大类，比如将城市聚为工业城市和农业城市，或将职员聚为高管和普通员工等。

聚类分析的基本规则是在相似的基础上分类，即将相似的观测值聚为一类，将不够相似的观测值分到不同的类别中。由于观测值的类别是未知的，因此聚类分析的结果非常自由，不对结果中类别的个数作出限制。与其他机器学习算法类似，聚类分析法使用训练数据训练模型，并将训练好的模型用于判断新观测值的类别。

聚类分析具有简单直观的特点，它的算法往往比较简单，在较复杂的数据集上也能以较高的效率输出结果，聚类分析能够发现数据中未知的信息，起到为决策者提供决策依据的作用。如今已有多种成熟的聚类分析方法流行于各个领域，其中较著名的有 KNN 聚类法、系统聚类法、动态聚类法、模糊聚类法等、基于密度的方法和基于网格的方法等。

聚类分析既可以对观测值进行聚类，也可以对变量进行聚类，当对观测值进行聚类时，聚类分析又称为 Q 型聚类；当对变量进行聚类时，聚类分析又称为 R 型聚类。

KNN 聚类分析是一种基于训练样本和新数据的属性分类的方法。KNN 聚类分析要求训练集中的样本数据已知，并要求每一类别中的观测值样本数目大致相同，否则结果将出现较大的偏差。

系统聚类法是另一种十分流行的方法，它首先将每个样本都视为单独的一类，然后根据样本之间的相似程度来两两合并，最终合并至我们指定的类别个数为止。与 KNN 聚类分析法相比，系统聚类法并不要求每个类别的个数相同，此外，无论观测值有多少个维度，系统聚类法都能画出聚类图，但 KNN 聚类法则不能直观表现聚类结果。

最后一类常用的聚类方法是快速聚类法，它又称为 K 均值法，与 KNN 聚类法不同，K 均值法并不知道训练集中观测值的类别，它以观测值之间的距离作为度量观测值相似程度的指标，使用迭代的方法训练模型，8.4 节讨论了它的细节。K 均值法与 KNN 十分相似，它们都具有计算简洁的优点，它们同样受聚类个数和初始分布的影响。

8.2 使用 R 实现 KNN 聚类

KNN 聚类是最简单的一种聚类方法，本节将讨论 KNN 聚类的思想和模型，并详细阐述 KNN 算法的优缺点。RR 提供了好几种实现 KNN 算法的函数，本节还将讨论这些函数的使用方法。

8.2.1 KNN 算法的思想和模型

KNN 算法也称为 K 近邻算法，像所有的机器学习算法一样，KNN 算法同时需要一份训练集和一份测试集。KNN 算法要求训练集中的观测值已经标出了类别变量和其他用于判断类别的非类别变量，而测试集中的观测值则不需要标出类别变量。尽管 KNN 算法需要一份训练集，但它实际上并不对训练集进行训练，而是直接拿来使用。

我们希望 KNN 算法能为测试集中的观测值预测类别。KNN 算法用非类别变量度量训练集中样本和测试集中样本的距离，并根据距离测试样本最近的 k 个训练样本的类别来判断测试样本的类别。

图 8.1 展示了 KNN 算法的思想。图中用红、黄、蓝三种空心小圆圈画出了 150 个类别已知的样本，显然，这些样本除类别变量外，还有 SW 和 SL 两个变量。不妨假设它们是一份训练集。KNN 算法在得到类别未知的新样本后，会根据新样本的 SW 和 SL 值比较它们与其他样本的距离。

图 8.1 中用两个星星标出了两个新样本，并用一个圆圈圈出了离它们最近的样本点。对于黑色的小星星来说，圆圈中圈住了三个黄色的样本点，这三个样本点显然是距离小黑星星最近的三个样本点，当圆圈放大后，圈住的样本点也将更多。由于距离小黑星星最近的三个样本点都是 *virginica* 类别的，因此 KNN 算法就会认为小黑星星也是 *virginica* 类别的。

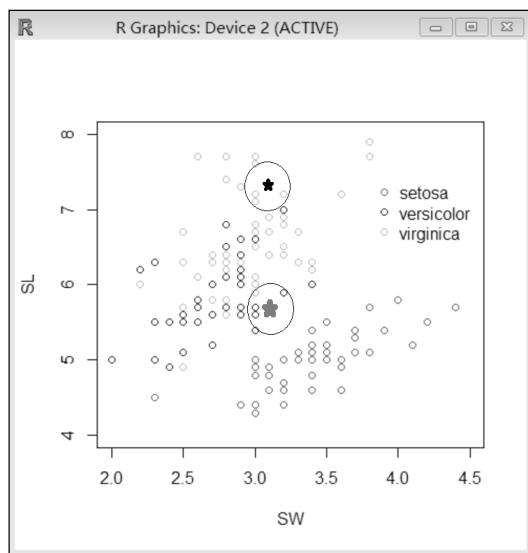


图 8.1 KNN 算法示意图

位置稍微靠下一些的大红星星同样圈了一个圈。这个以大红星星为中心的圆圈内有 4 个样本点，其中三个是蓝色的，一个是黄色的。由于蓝色样本多于黄色样本，因此 KNN 算法会认为大红星星是 `versicolor` 类别的。

在图 8.1 中，度量新样本值和训练样本的距离的方法显然是欧式距离法，也就是以两个样本的直线距离作为比较标准，欧氏距离在训练样本呈椭圆分布时工作良好，但当训练样本呈现内凹或外凸的奇怪形状时，根据样本密度计算距离更加合适。

图 8.1 只是一个非常简单的例子，在实际算法中，KNN 算法同样能够解决样本的非类别变量较多的问题，只是当样本中的非类别变量多于三个时，将无法进行数据可视化，也无法直观地观察数据的分布状况。此外，KNN 算法也未必会以新样本的距离小于某个固定值的训练样本来推测新样本的类别，更合理的方法是使用距离训练样本最近的几个训练样本，这样可以避免出现新样本过于离群，在固定距离内找不到训练样本的尴尬。

最后一点需要说明的是，KNN 算法显然假设每个类别的训练样本数目大致相同。当有些类别中的训练样本数据过少，而另一些类别中的训练样本数据过多时，那么新样本属于训练样本较多的类别的概率就会增大，这有时是不合理的。

8.2.2 使用 R 实现 KNN 聚类

程序包 `class` 收集了与 KNN 聚类相关的三种函数，其中用起来最方便的是 `knn.cv()` 函数，不妨仍以 `iris` 数据集为聚类对象，对其实现 KNN 聚类方法。

```
> library(class)
> sam <- sample(c(1:150),100)
> iris[6] <- c(rep(0,50),rep(1,50),rep(2,50))
> iris.sam <- iris[sam,]
> iris.test <- iris[-sam,]
```

上述代码首先加载了 `class` 数据集，然后使用 `sample()` 函数抽取了一列随机数，`sample()` 函数中的第一个参数指定被抽取的对象是数列 1~150，第二个参数指定抽取 100 个随机数赋给 `sam`。第 3 列代码生成了一个长度为 150 的向量，并令它作为 `iris` 的第 6 列，这列变量由 50 个 0、50 个 1 和 50 个 2 组成，分别对应 `iris` 中的三种类别，我们希望在后续代码中使用它来代表 `iris` 的类别，以便于简化代码。

上述代码中的最后两行代码生成了一个训练集和一个测试集。训练集 `iris.sam` 由 `iris` 中行数为 `sam` 中随机数的行组成，一共有 100 个观测值，测试集 `iris.test` 则由剩下的 50 个观测值组成。注意，输入代码时要在 `sam` 和 `-sam` 后加上逗号，表示筛选的是 `iris` 中的行，否则 R 将默认筛选的是 `iris` 中的列，而 `iris` 只有 6 列，因此显然会报错。

```
> plot(iris.sam[1:2],col=iris.sam$Species)
```

上述代码使用 `plot()` 函数查看了训练集样本的真实类别，其中的第一个参数指定绘图对象是 `iris` 的前两列变量 `Sepal.Length` 和 `Sepal.Width`，第二个参数则指定使用 `iris` 中的 `Species` 变量标出散点的颜色。

图 8.2 是 R 的返回结果，其中黑色的散点与其他散点的距离明显较远，有很清楚的

分界线，但红色散点与绿色散点则混杂在一起，难以区分。不妨试一试 KNN 算法，能够正确地为本样本分类。

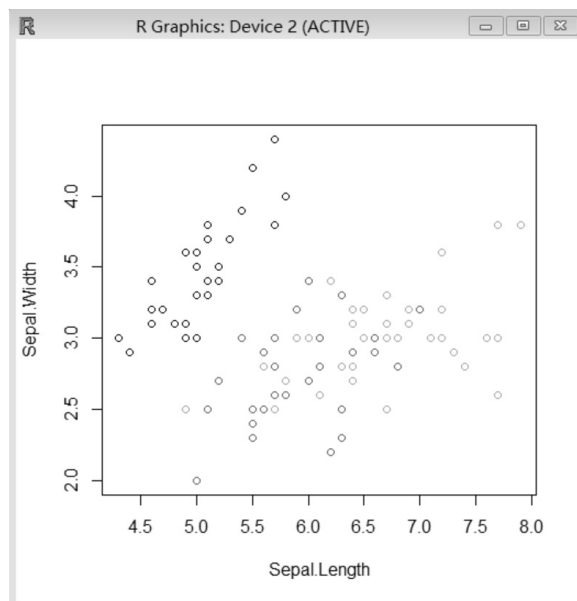


图 8.2 训练集样本的真实类别

```
> knn.cv(iris.sam[,1:2],cl=iris.sam[,6],k=10)
[1] 2 1 2 2 1 1 1 0 1 0 2 0 1 2 0 2 0 2 0 2 2 2 1 0 1 2 1 0 0 2 1 0 1 2 2
[37] 2 2 0 1 1 1 1 1 1 0 0 2 1 2 2 0 0 1 1 0 2 1 1 0 0 0 2 2 0 1 2 0 2 1 1 1
[73] 2 2 1 0 0 2 2 1 2 1 2 1 2 2 0 1 1 2 1 2 2 2 0 0 0 2 0 2
Levels: 0 1 2
> kn<-knn.cv(iris.sam[,1:2],cl=iris.sam[,5],k=10)
> table(iris.sam$Species,kn)
      kn
      0  1  2
setosa  29  0  0
versicolor  0 22 12
virginica   0 11 26
> plot(iris.sam[1:2],col=kn)
```

`knn.cv()` 算法是 `class` 包提供的专门用于构建 KNN 模型的函数。上述代码首先使用 `knn.cv()` 算法构建了一个 KNN 模型。代码一共指定了三个参数，第一个参数指定样本数据是 `iris.sam` 的前两列；第二个参数指定 `cl` 为 `iris.sam` 的第 6 列，即使用 `iris.sam` 的第 6 列作为样本数据的类别标签；第三个参数 k 为 10，即对每个新样本来说，使用离它们最近的 10 个训练样本的类别来判断它们的类别。

由于第 1 条代码并未指定测试集，因此 `knn.cv()` 函数自动将训练集作为测试集，计算了训练集中 100 条观测值的预测类别。R 直接返回了这 100 个数据的预测类别，为了便于查看，第 2 行代码将 KNN 模型结果赋给了变量 `kn`，并使用第 3 行代码查看了 `iris.sam` 中 `Species` 变量和 `kn` 变量的列联表，显然，属于 `setosa` 的 29 个样本并未被分错类别，

但属于 *versicolor* 的 34 个样本却有 12 个被分到 *virginica* 类别中，属于 *virginica* 的 37 个样本也有 11 个被分到 *versicolor* 类别中。最后一行代码使用 `plot()` 函数查看了 KNN 算法的最终预测结果。

图 8.3 是上述代码中 `plot()` 函数的返回结果，与图 8.2 相比，图 8.3 中的样本点被均匀地分成了三块，红色样本和绿色样本不再有交叉，但这显然与真实类别不一致。增加样本个数有利于改善这一情况。此外，`knn.cv()` 函数也提供了其他度量距离的参数，不妨逐一试一试。

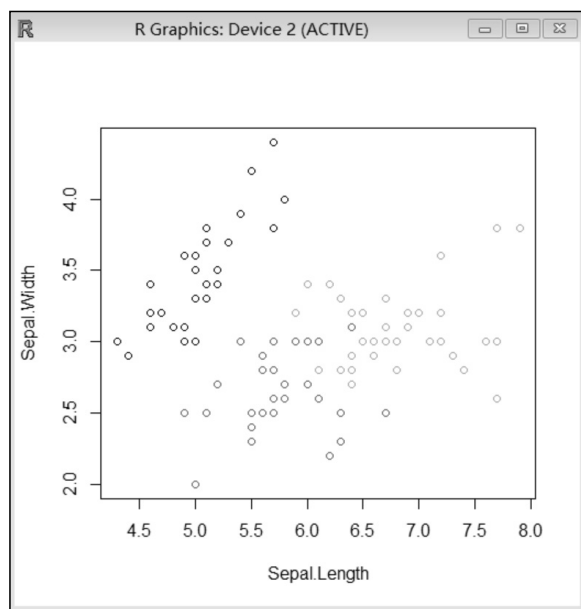


图 8.3 训练集样本的预测类别

8.3 使用 R 实现系统聚类

本节将介绍系统聚类的相关知识。系统聚类同样基于相似度对样本进行聚类，但其思想与 KNN 算法截然不同。本节将比较系统聚类与 KNN 聚类的异同，还将讨论如何使用 R 实现系统聚类。

8.3.1 系统聚类的思想和模型

系统聚类法又称为层次聚类法，其算法思想与 KNN 算法有很大不同。KNN 算法的训练样本的类别是已知的，它所做的工作实际上是将新样本分到已知的类别中去，样本数据的类别是固定的，但系统聚类法所要做的则是根据样本信息归纳出类别来，在模型创建完毕之前，样本的类别是未知的。另外，KNN 算法区分训练集和测试集，而系统聚类法则不区分训练集和测试集，自始至终只在一个数据集上创建模型并获得最终结论。

在初始状态，系统聚类法认为每一个样本都是单独的一类，含 n 个样本的数据集就

有 n 个类别。通过计算类别与类别之间的距离，系统聚类法会将间隔最小的两类数据合并为一个新的类，这时数据集中的类别数就减少为 $n-1$ 个类。进一步的，系统聚类法再次找出间隔最小的两类数据，再次将它们合并为一个类，这时数据集中的类别数将进一步减少为 $n-2$ 个类。以此类推，最终数据集中将只剩下一个大类，即包含全体数据的类。

显然，构建系统聚类模型的重点在于如何构建度量类别之间距离的公式。在最初时，每个类别都只有一个样本，度量样本与样本之间的欧氏距离即可，但当第一次合并完成，出现包含多个样本的类别时，度量类别与类别之间的距离就会显得复杂。实际上，不同的度量方式正是区分不同系统聚类方法的主要指标。

最小距离法比较了两个类别中彼此挨得最近的两个样本，它们度量了两个类别的最小距离；最大距离法比较了两个类别中彼此挨得最远的两个样本，它们度量了两个类别的最大距离。这两种方法都仅运用了每个类别中一部分样本的信息，是较粗糙的方法。

类平均法综合了最小距离法和最大距离法，它比较了两个类别之间每一对样本对的平均距离，假设类别 X 有 i 个样本，类别 Y 有 j 个样本，则这两个类别能构造出 $i \times j$ 个样本对，两个类别间的平均距离就是这 $i \times j$ 个样本对的平均距离。

最后一类常用的方法是离差平方法。它利用类别中数据的分散程度度量两个类别的距离。假设类别 X 和类别 Y 会合并为类别 Z ，则类别 Z 的离差平方和减去类别 X 的离差平方和后再减去类别 Y 的离差平方和就是类别 X 和类别 Y 的离差距离。

类平均法和离差平方法都利用类别中的全体信息进行推断，因此比最小距离法和最大距离法更加合理。其中，离差平方方法的准确度又要比类平均法更高，通常情况下是最佳的系统聚类方法。

8.3.2 使用 R 实现系统聚类

R 的基础包提供了用于构建系统聚类模型的 `hclust()` 函数，只需直接调用即可。本节仍使用 8.2 节中使用随机数给出的长度为 100 的训练集 `iris.sam` 作为原始数据，并尝试为这 100 条样本进行分类。

```
> disma <- dist(iris.sam[1:4],method="euclidean")
> head(disma)
[1] 0.5385165 0.5099020 0.6480741 0.1414214 0.6164414 0.519615
> clur <- hclust(d=disma,method="ward.D")
> plot(clur,labels=iris.sam[,6])
```

上述代码首先使用 `dist()` 函数为 `iris.sam` 的前 4 列创建了一个距离矩阵 `disma`，`dist()` 函数的第二个参数指定计算距离的方法为 `euclidean`，即使用欧氏距离度量样本之间的距离。第 2 行代码使用 `head()` 函数查看了 `disma` 的前 6 个元素，显然，这是一个数值类型的矩阵。

第 3 行代码使用 `hclust()` 函数构建了系统聚类模型，`hclust()` 函数的应用对象为距离矩阵，它的第一个参数指定被应用的对象是 `disma`；第二个参数 `method` 指定为 `ward.D`，即使用离差平方法度量类与类之间的距离。

创建好模型后，我们并未直接查看模型中的结果，而是将模型结果赋给了 `clur` 变量。观察系统聚类结果的最佳方式是根据树状图查看聚类模型中的信息，上述代码的最后一行使用 `plot()` 函数绘出了树状图，由于 `clur` 是系统聚类的结果，因此 `plot()` 直接绘出了树状图，而 `plot()` 函数的第二个参数 `labels` 则指定使用 `iris.sam` 的第六列变量作为树状图的标签。

图 8.4 给出了系统聚类的树状图，该树状图从每一个单独的样本开始，三三两两地逐渐合并为一些小的类别，这些小的类别继续慢慢向上合并，最终合并为一个整的大类。将两个小类别连接为一个大大类的竖线有长有短，竖线越长，就说明被合并的两个类别越不相似；竖线越短，就说明被合并的两个类别越相似。

在树状图的最低端标出了每一个样本对应的类别。我们在 8.2 节中创建 `iris[6]` 的作用在这里体现了出来，如此小的位置上难以挤下 `Species` 提供的类别名，而类别 0、1、2 则能勉强挤进去。观察这些样本的对应值，所有类别为 0 的样本都集中在树状图的左端；而类别为 2 的样本大部分集中在树状图的中间，还有一小部分处于树状图的右侧，将类别为 1 的样本一分为二，甚至还有一个异常值单独藏在类别为 1 的样本中。如果不在 `plot()` 函数中设置 `labels` 参数，树状图的底部将标注每一个样本的名称。

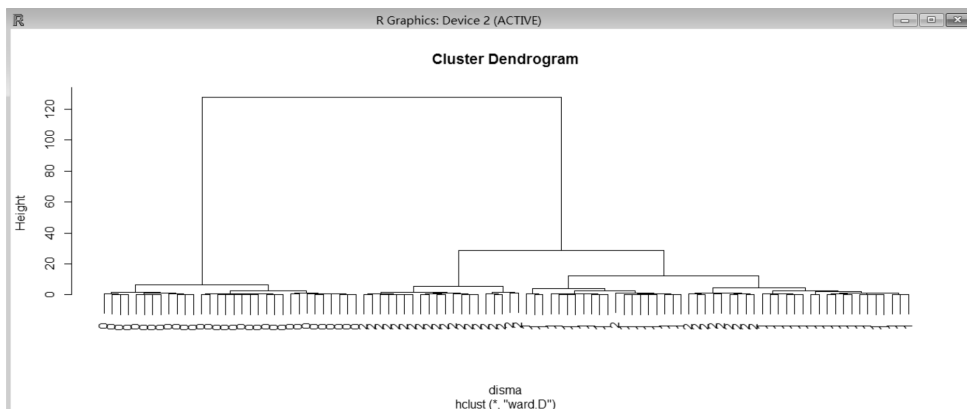


图 8.4 使用树状图查看系统聚类的结果

如果不考虑 `Species` 变量提供的 `iris` 类别信息，仅从树状图来看，将 `iris` 数据集分为两类、三类或四类都具有一定的合理性。碎石图在样本究竟该分为多少类这一问题起到辅助作用。

```
> plot(clur$height)
```

在系统聚类模型的结果中有一个 `height` 分量记录了类别间的相似程度，图 8.4 正是以 `height` 作为纵轴。使用 `plot()` 函数为 `height` 单独绘图，即可得到系统聚类的碎石图。

图 8.5 是上述 `plot()` 函数返回的碎石图。该碎石图由许多小圆圈组成，它们缀连成一条曲线，看起来就像一些碎石子一样。在图 8.5 的右侧，有三个碎石远离其他碎石，独立地待着。从右往左看，第 1 块碎石到第 2 块碎石的角度十分陡峭，而从第 4 块碎石开始，石子的走向变得平稳起来，结合前三块碎石远离其他石子的情况，显然，将 `iris.sam` 分为三类是合适的，查看图 8.4 的树状图，每一个类别的成员也能标注出来。

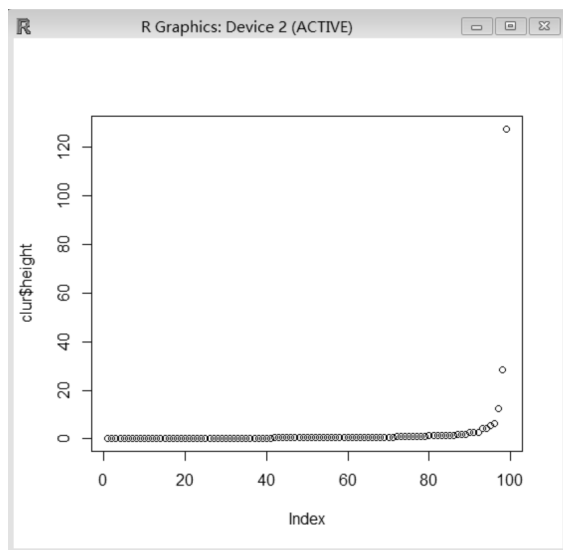


图 8.5 为系统聚类模型绘制碎石图

8.4 使用 R 实现快速聚类

快速聚类又称为 K 均值聚类，是在 KNN 聚类模型上进一步发展得到的一种算法。本节将讨论快速聚类的思想，使用 R 创建快速聚类模型，并对比了 KNN 模型和快速聚类模型的分类效果。

8.4.1 快速聚类的思想和模型

快速聚类是在 KNN 聚类思想的基础上进一步深入得到的模型。KNN 聚类分析假设测试样本的类型是已知的，它略过了训练样本的部分，而快速聚类则假设测试样本的类型是未知的，在 KNN 聚类思想的基础上添加了样本的训练过程。

初始状态时，快速聚类需要数据分析师手动指定样本的类别个数，即打算将样本分为几个类。在得到样本类型个数后，快速聚类首先会给每个样本随机指定一个类别，并根据这个随机类别计算出每个类别的中心位置。

由于样本的类别是随机指定的，因此此时样本点散漫地分布在空间中，其中必然存在一些样本点距离其他类别的中心位置要比距离自己被随机分配的类别的中心位置更加近，即有一些样本点显然被分错了类。修改这些分错类的样本点的类别，此时每一个类别中包含的样本点就发生了变化。然后快速聚类会根据每个类别中包含的新样本计算新的中心位置。

有了新的中心位置后再一次检查是否有样本点分错了类，再次修改样本点的类别，再次计算新的中心位置，直至每个类别的中心位置都落到合理的位置，每个类别的成员都不再变动为止。

图 8.6 展示了快速聚类中每个类别的中心位置是如何随着迭代次数的增加而发生改

变的。图 8.6 中用 4 个黑色星星标注了在 4 次迭代过程中绿色圆圈的中心位置的改变，箭头指出了中心位置移动的方向。随着迭代次数的增加，每个类型所包含的样本点趋于稳定，中心位置的移动幅度也会变小，此外，中心位置有可能在最终位置附近出现徘徊，但快速聚类一般都会收敛到一个最优值处。

也许你还没有意识到，但图 8.6 已经显现出了快速聚类的缺点。快速聚类与 KNN 聚类相似，只适用于每个类别的样本个数大致相同的情况。此外，快速聚类的最终结果与每个类型的中心位置初始状态有关，即当初始状态不同时，快速聚类有可能得到不同的结果。考虑到快速聚类总是为每个样本随机分配初始类别，因此每一次快速聚类的结果都可能存在微小差异。解决该问题的最佳方法是为同一份数据多做几次快速聚类。

此外，快速聚类需要数据分析师手动指定类别个数，类别个数的设置对最终聚类结果的影响非常大，而我们确定类别个数的最简单办法就是看一看原始数据大概的分布，再依据分布指定聚类个数，而散点图是查看数据分布的最佳图形，这就意味着当原始数据超过三维时，确定类别个数将变得有些困难。此时数据分析师不得不借助其他的先验知识为快速聚类指定聚类个数。

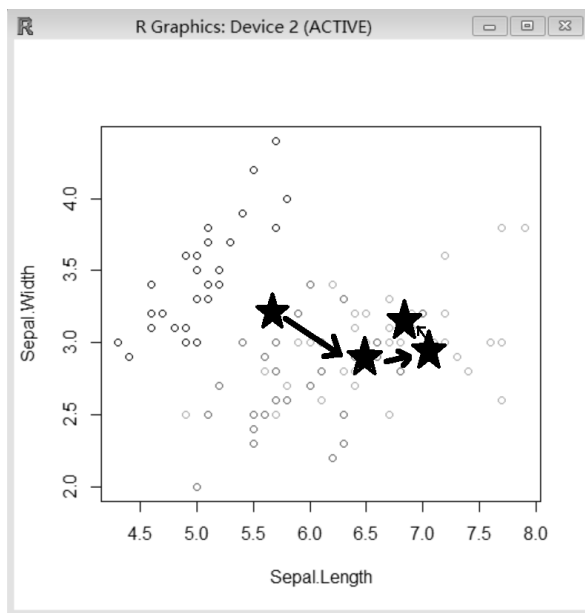


图 8.6 快速聚类的迭代过程

8.4.2 使用 R 实现快速聚类

R 中的基础包同样提供了用于构建快速聚类模型的 `kmeans()` 函数。无须加载包，直接调用该函数即可构建快速聚类模型。

```
> kc <- kmeans(iris.sam[1:2],3)
> kc
K-means clustering with 3 clusters of sizes 33, 24, 43
```

```

Cluster means:
  Sepal.Length Sepal.Width
1      5.045455      3.369697
2      7.075000      3.087500
3      5.976744      2.800000

Clustering vector:
72 143 126 52 61 65 99 30 69 5 101 19 138 130 7 71 40 110 28
3 3 2 3 3 3 1 1 3 1 3 1 3 2 1 3 1 2 1
20 131 112 59 135 49 73 136 88 14 22 121 62 4 56 145 75 77 141
1 2 3 2 3 1 3 2 3 1 1 2 3 1 3 2 3 2 2
44 70 128 150 93 80 139 41 15 118 60 116 119 24 16 81 114 29 86
1 3 3 3 3 3 3 1 1 2 1 3 2 1 1 3 3 1 3
90 68 31 26 50 57 103 10 109 76 9 117 85 84 102 133 74 54 34
3 3 1 1 1 3 2 1 2 2 1 3 1 3 3 3 3 3 1
6 134 132 92 53 82 146 96 105 149 2 100 107 111 122 142 108 113 47
1 3 2 3 2 3 2 3 3 3 1 3 1 2 3 2 2 2 1
38 48 106 35 51
1 1 2 1 2

Within cluster sum of squares by cluster:
[1] 10.351515 6.171250 9.616744
(between_SS / total_SS = 70.8 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"
[5] "tot.withinss" "betweenss"    "size"         "iter"
[9] "ifault"

```

上述代码使用 `kmeans()` 函数为 `iris.sam` 构建了快速聚类模型，`kmeans()` 函数中的第一个参数指定模型的原始数据为 `iris.sam` 的前两列，第二个参数则指定将原始数据聚为三类。聚类结果存放在 `kc` 中，查看 `kc`，R 返回了一长串列表。

列表的第 1 行是对这个模型的概述，`kc` 所存放的模型是一个有三个分类的快速聚类模型，每个分类的具体样本个数为 33、24 和 43。列表的第二个元素 `Cluster means` 则给出了三个分类的中心位置坐标，将这三组坐标与图 8.7 相对照，可知，显然类别 1 为黑色圆圈，类别 2 为红色圆圈，类别 3 为绿色圆圈。列表的第三个元素 `Clustering vector` 则给出了快速聚类模型为 100 个样本计算出的最终类别。

列表的第 4 个元素给出了三个数值：10.351 515、6.171 250 和 9.616 744，它们分别是所有聚类变量的离差平方差之和、每个类别所有聚类变量的离差平方差之和，以及每个类别所有聚类变量的离差平方差之和的和，这三个指标又与列表第 5 个元素中的“`totss`”、“`withinss`”、“`tot.withinss`”相对应。而 `between_SS / total_SS` 给出的则是各类别间的聚类变量离差平方和之和。这些指标分别度量了聚类结果中每个类的内部，以及类与类之间的分散程度。

列表的第 5 个元素给出了快速聚类模型中各指标的名称，使用“快速聚类模型名+\$+指标名称”的格式即可方便地调用快速聚类模型中的指标。

```
> table(iris.sam$Species,kc$cluster)

      1  2  3
setosa 29  0  0
versicolor 3  5 26
virginica 1 19 17
```

上述代码使用 `table` 函数查看了 `iris.sam` 中 `Species` 变量与 `kc` 中 `cluster` 指标形成的列联表。显然，`setosa` 类型全部分入了第一类，并未分错类型；`versicolor` 类型主要分入了第三类，但有三个样本被分入第一类，有 5 个样本被分入第二类；`virginica` 类型有 19 个分入了第二类，有 17 个分入了第三类，还有一个分入了第一类。与 KNN 聚类分析的结果相比，快速聚类分析的结果显然要差一些。

```
> plot(iris.sam[1:2],col=kc$cluster)
```

上述代码查看了快速聚类的最终结果，`plot()` 函数的第一个参数指定画图对象为 `iris.sam` 的前两列变量；第二个参数 `col` 则指定使用 `kc` 中的 `cluster` 变量为散点着色。其结果如图 8.7 所示。

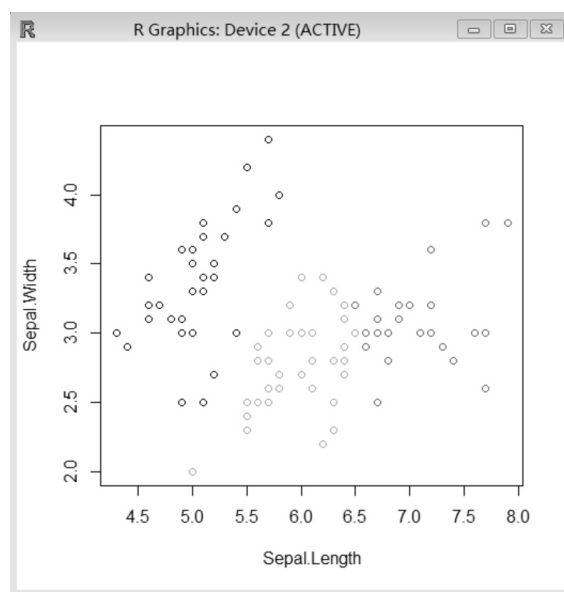


图 8.7 快速聚类的最终结果

观察图 8.7，散点仍旧被聚为黑色、绿色和红色三种颜色，与 KNN 聚类分析的结果相比，图 8.7 所示的分类更接近三个圆形结构。本节与 8.2 节在创建模型时使用了同样的原始数据和同样的距离度量方法，但 KNN 聚类中原始数据的类别信息是已知的，而在快速聚类中则未知。先验知识的不足是快速聚类效果不如 KNN 聚类的主要原因之一。

8.5 几种判别分析模型综述

判别分析是一种与聚类分析十分相似的统计分析方法，它与 KNN 聚类模型一样，都要求样本数据的类别是已知的，并使用已知的样本信息推断新样本的类别。本节将介绍两种常见的判别分析，并讨论它们的思想与应用。

8.5.1 距离判别模型

距离判别模型是最简单的一种判别分析模型。在有关 KNN 模型的介绍中可以发现，KNN 模型利用离新样本最近的 k 个训练样本的类别判断新样本的类别，这与距离判别模型的思想十分相似。KNN 模型虽然称为聚类模型，但实质上它是一种分类模型，即测试类别已知，不需要手动指定样本类别。

判别分析是分类模型中最典型的一种模型，所有的判别分析都要求样本类别已知。距离判别法与 KNN 聚类模型的思想较为相似，只不过 KNN 用 k 个训练样本推测新样本的类别，而距离判别分析则用每个类别的中心坐标推测新样本的类别。

```
> S0<-subset(iris.sam[,1:4],iris.sam[,6]==0)
> S1<-subset(iris.sam[,1:4],iris.sam[,6]==1)
> S2<-subset(iris.sam[,1:4],iris.sam[,6]==2)
> center0<-colMeans(S0)
> center1<-colMeans(S1)
> center2<-colMeans(S2)
> center<-rbind(center0,center1,center2)
> center
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
center0	5.027586	3.431034	1.486207	0.262069
center1	6.027778	2.791667	4.275000	1.325000
center2	6.508571	2.962857	5.502857	2.008571

仍以前几个小节中使用的 `iris.sam` 数据集作为训练集，`iris.test` 数据集作为测试集。上述代码首先使用三行 `subset()` 函数从 `iris.sam` 中提出了子集，`subset()` 函数的第一个参数指定提取的部分为 `iris.sam` 的前 4 列，第二个参数指定了提取的条件。这三行代码将 `iris.sam` 中第 6 列变量中值为 0、1、2 的行分别提取出来放到了 `S0`、`S1`、`S2` 中。此时 `S0`、`S1`、`S2` 中分别存放了类别为 `setosa`、`versicolor`、`virginica` 的样本的花萼长、花萼宽、花瓣长和花瓣宽。

接下来的 4 行代码使用 `colMeans()` 函数对 `S0`、`S1`、`S2` 分别按列求均值，并使用 `rbind()` 函数将这些均值按行连接成一个矩阵。命令 `center` 查看了其中存储的信息，这是一个 3 行 4 列的矩阵，每行代表一个类别，每列代表一个变量。矩阵中的 12 个数值给出了三组坐标。

距离判别法使用每一类别的列均值度量每一类的中心位置，因此 `center` 中第一行数值给出的坐标（5.027 586，3.431 034，1.4 862 077，0.262 069）即为类别为 `setosa` 的测试样本的中心位置，类似地，`versicolor` 类别与 `virginica` 类别的中心位置也存放在矩阵 `center` 中。我们希望同时创建存储了每个类别的中心位置坐标的向量及存储了全部类别

的中心位置坐标的矩阵，在后续程序中将会分别调用它们。

```
> plot(iris.sam[1:2],col=iris.sam$Species)
> points(center[,1],center[,2],pch=3,cex=2)
```

上述代码首先使用 `plot()` 函数绘出了 `iris.sam` 的前两列，并将 `col` 参数设置为 `Species`，即使用 `Species` 中的类别信息区分它们的颜色。利用 `points()` 函数在 `plot()` 绘出的图像中添加了三个点，这三个点的坐标由 `center` 的第一列和第二列给出，`pch` 参数为 3，`cex` 参数为 2，即绘制类型为第 3 类、大小为标准大小的两倍大的点。图 8.8 中的三个黑色十字符号标出了这三个点。

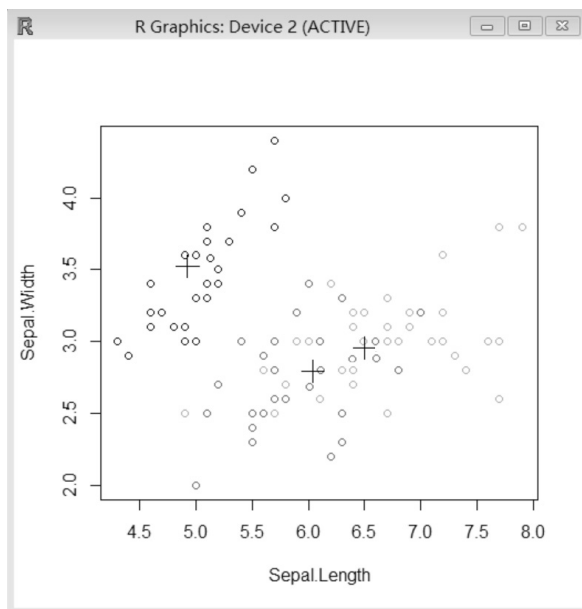


图 8.8 为 `iris.sam` 数据集绘制中心位置

这三个十字符号标出了 `iris.sam` 中三种类别的中心位置。当新样本进入模型后，模型会比较新样本与这三个中心坐标的距离，并将新样本归入离它最近的中心坐标所代表的类别中。图 8.8 只给出了二维平面上的中心位置，`center` 中存储的中心位置共有 4 个维度，即每个类别在四维空间中的中心位置。

```
> C0<-(length(S0[,1])-1)*cov(S0)
> C1<-(length(S1[,1])-1)*cov(S1)
> C2<-(length(S2[,1])-1)*cov(S2)
> C<-(C0+C1+C2)/length(iris.sam[,1])-3
> C
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	0.24473943	0.09394289	0.14398357	0.03576015
Sepal.Width	0.09394289	0.12095684	0.05775224	0.03853961
Petal.Length	0.14398357	0.05775224	0.15054144	0.03999788
Petal.Width	0.03576015	0.03853961	0.03999788	0.04462380

上述代码计算了每个类别中测试样本的自由度与协方差阵之积 C0、C1、C2，并计算了三个类别中测试样本的合并协方差阵 C。观察 R 的返回结果，C 中存储了一个对称的方阵，接下来要利用合并协方差阵 C 计算新样本与每个类别中心位置的马氏距离。

```
> for (i in 1:length(iris.test[,1])) {
+ R0 <- mahalanobis(iris.test[i,1:4],center=center0,cov=C)
+ R1 <- mahalanobis(iris.test[i,1:4],center=center1,cov=C)
+ R2 <- mahalanobis(iris.test[i,1:4],center=center2,cov=C)
+ c <- c(R0,R1,R2)
+ if (R0==min(c)){iris.test[i,6] <- 0}
+ else if (R1==min(c)){iris.test[i,6] <- 1}
+ else {iris.test[i,6] <- 2}
+ }
> head(iris.test)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species V6
1          4.9          3.0          1.4          0.2  setosa  0
2          4.7          3.2          1.3          0.2  setosa  0
3          4.6          3.4          1.4          0.3  setosa  0
4          5.0          3.4          1.5          0.2  setosa  0
5          4.9          3.1          1.5          0.1  setosa  0
6          4.3          3.0          1.1          0.1  setosa  0
```

上述代码使用 for 循环判断了 iris.test 中每个样本的类别。for 循环的循环名指明循环范围为 iris.test 的全部行，由于 length() 函数只能作用于向量，因此这里使用 length() 函数计算了 iris.test 中第 1 列的长度。for 循环的循环体使用三行 mahalanobis() 函数计算了 iris.test 数据集中第 i 行样本与三个类别中心坐标的马氏距离，mahalanobis() 函数的第一个参数表明被计算的第一个对象是 iris.test 数据集第 i 个样本的前 4 个变量值，第二个参数表明被计算的第二个对象是 center0、center1 和 center2，第三个参数则表明使用协方差阵 C 计算这二者的距离。

循环体中第 4 行代码创建了一个向量 c，存储了三个距离值。接下来的两组 ifelse 语句用于判断 R0、R1、R2 与 c 中最小值的关系，当 R0 为 c 中最小值时，新样本即为第 0 类；当 R1 为 c 中最小值时，新样本即为第 1 类；当 R2 为 c 中最小值时，新样本即为第 2 类。判别分析预测的类别存储在 iris.test 的第 6 列中，使用 head() 函数查看 iris.test，此时数据框中已增加了一列变量 V6。对比 iris.test 中 V6 与 Species 的类别信息的差异，即可得知距离判别分析的正确率。

8.5.2 Fisher 判别模型

Fisher 判别分析也称为典型判别，是另一类常用的判别分析。它使用了降维的思想来处理数据，即将高维空间中的数据投影到低维空间中，并根据低维空间中训练样本的分布为新样本分类。为数据降维能够大幅简化计算，而恰当的低维空间则可以尽可能多地保留原始信息。

R 中的 MASS 包提供了用于进行 Fisher 判别分析的 lda() 函数，仍以 iris.sam 数据集

作为判别分析的对象。

```
> library(MASS)
> dis<-lda(Species~Sepal.Length+Sepal.Width+Petal.Length+Petal.Width,iris.
sam)
> dis
Call:
lda(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
    data = iris.sam)

Prior probabilities of groups:
      setosa versicolor  virginica
      0.29      0.36      0.35

Group means:
      Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa           5.027586    3.431034      1.486207    0.262069
versicolor       6.027778    2.791667      4.275000    1.325000
virginica         6.508571    2.962857      5.502857    2.008571

Coefficients of linear discriminants:
      LD1      LD2
Sepal.Length  0.631378  1.02509346
Sepal.Width   2.054789 -2.81486921
Petal.Length -2.015069  0.04928358
Petal.Width  -3.244798 -1.88389129

Proportion of trace:
      LD1      LD2
0.9908 0.0092
```

上述代码首先加载了 MASS 包，然后使用 `lda()` 函数构建了 Fisher 判别分析模型。`lda()` 函数中的第一个参数指定 `Species` 为类别变量，`Sepal.Length`、`Sepal.Width`、`Petal.Length`、`Petal.Width` 为用于预测类别的变量；第二个参数则指定 `iris.sam` 为被应用的数据框。`lda()` 函数的书写格式与用于方差分析的 `aov()` 函数较为相似，当数据框中除类别变量以外的其他变量都是用于预测类别的变量时，也可简写为“类别变量~.”的形式，符号“.”代表除类别变量外的全部变量。

Fisher 判别分析的结果存储在 `dis` 中，查看 `dis`，这是一个存储了 5 个元素的列表。列表中第一个元素 `Call` 给出了判别分析模型的具体信息，包括模型中的类别变量、非类别变量和数据来源等；第二个元素 `Prior probabilities of groups` 给出了新样本信息未知时，属于这三个类别的先验概率，这一概率与每种类别中训练样本的个数有关，也与每种类别中训练样本的密集程度有关。

第三个元素给出了训练样本中心坐标，这三个中心坐标与距离判别中 `center` 给出的中心坐标相同；第 4 个元素则给出了 Fisher 判别分析给出的空间转换系数，在 `iris.sam` 构建的模型中只给出了两组系数，因此 Fisher 判别分析将 `Sepal.Length`、`Sepal.Width`、

Petal.Length、Petal.Width 四个维度投射到了一个二维空间中，该二维空间的两个维度分别为 LD1 和 LD2。

Coefficients of linear discriminants 同样给出了系数转换公式，从原始空间映射到 LD1 维的转换公式为 $LD1=0.631\ 378\times SL+2.054\ 789\times SW-2.015\ 069\times PL-3.244\ 798\times PW$ ，其中使用 SL、SW、PL、PW 作为 Sepal.Length、Sepal.Width、Petal.Length、Petal.Width 的简写。类似地，从原始空间映射到 LD2 的转换公式为 $LD2=1.02\ 509\ 346\times SL-2.81\ 486\ 921\times SW+0.04\ 928\ 358\times PL-1.88\ 389\ 129\times PW$ 。根据这两个映射公式，训练样本即可投影到新的二维空间中，训练样本的三个类中心坐标也能投影到新的二维空间中。

当新样本进入模型后，Fisher 判别分析会自动计算出新样本的 LD1 值和 LD2 值，并比较新样本的二维坐标与三个类中心的二维坐标的距离，从而为新样本预测类别。新样本的 LD1 值和 LD2 值也称为新样本在 LD1 维和 LD2 维上的综合得分。Proportion of trace 元素给出了 LD1 和 LD2 的信息容量，LD1 承载了原始数据中 99.08% 的信息，LD2 则仅承载了原始数据中 0.92% 的信息。该元素提供了取舍 Fisher 判别模型中新维度的标准。

```
> p.dis<-predict(dis)
> c<-cbind(iris.sam$Species,p.dis$x,p.dis$class)
> head(c)
```

		LD1	LD2	
120	3	-4.676876	2.04714391	3
132	3	-4.633090	-1.38191789	3
125	3	-5.332070	-1.42748307	3
32	1	8.032063	-0.04596732	1
9	1	7.223757	0.70822372	1
64	2	-2.246385	0.35284886	2

上述代码使用 predict() 函数预测了判别分析模型 dis 的结果，由于并未给 predict() 函数指定第二个参数，因此，predict() 函数默认将 dis 模型的训练样本视为测试样本，为其重新预测分类。将 iris.sam 中的变量 Species、p.dis 中的 x 变量、p.dis 中的 class 变量按列连接起来赋给变量 c，并查看 c 的前 6 行。由 R 的返回结果可知，iris.sam 中的 Species 变量存储了训练样本的真实类别，p.dis 中的 class 变量则存储了训练样本的预测类别，而 p.dis 中的 x 变量则存储了训练样本在维度 LD1、LD2 下的新坐标。

对比 c 中第 1 列和第 3 列的结果，即可得知 Fisher 判别分析模型 dis 的准确度。实际上，Fisher 判别分析和距离判别分析的准确度都是百分之百，这与这两个模型总共使用了 4 个变量来预测测试样本的类别有关，如果在之前的三种聚类分析中使用 4 个变量来预测样本类别，模型的准确度同样也会大幅提高。

第 9 章 R 中的主成分分析和因子分析

本章的主题是降维。主成分分析和因子分析是最经典的降维算法。本节将介绍主成分分析和因子分析的模型与应用，并比较这两者的异同。在本章的最后，将演示主成分分析和因子分析的结果是如何和回归分析、聚类分析联系起来的。

9.1 主成分分析的实现与应用

本节将介绍主成分分析的思想模型与 R 中的相关函数，并讨论如何解读主成分分析中的主要指标。主成分分析是较为简单的一种降维方法，本节有关主成分分析的讨论同样适用于因子分析。

9.1.1 主成分分析的模型假设和数据处理

在第 6 章有关回归分析的介绍中已经提到，回归分析要求自变量和因变量都服从正态分布，且自变量之间相互独立。在现实生活中，这两条假设一般不可能完全满足，首先，原始数据一般不会完全服从正态分布，其次，自变量之间往往存在相关性。

第一个问题一般可以通过对原始数据取对数或删掉一些异常值来解决，而第二个问题则比较严重。以社会问题或经济问题为例，一个因变量往往需要使用十几个自变量来度量，而自变量的个数越多，自变量之间存在相关性的可能性就越高。此外，过多的自变量也会增加计算的复杂度，降低模型效率。

在判别分析中已经提到了投影的概念，这一概念同样适用于降维问题。既然十几个自变量之间存在相关关系，也就是不同的自变量隐含着同样的信息，如果能将比较类似的自变量投影到同一个维度上，就可以达到同时消除自变量之间的多重共线性，又降低计算复杂度的效果。

主成分分析就是这样一种专门将高维数据投影到低维空间的分析方法。显然，主成分分析要求被分析的变量之间具有相关性，否则主成分分析将失去原有的意义；其次，主成分分析假设变量均服从正态分布，与回归分析类似，这条假设要求并不是特别严格。

R 中自带的数据集包含一份有关瑞士城市发展指标的数据 `swiss`，它存储了来自 47 个城市的 6 个指标，下面不妨看一看这份数据是否适合做主成分分析。

```
> head(swiss)
      Fertility Agriculture Examination Education Catholic Infant
.Mortality
Courtelary    80.2         17.0         15         12         9.96        22.2
Delemont      83.1         45.1          6          9        84.84        22.2
```

Franches-Mnt	92.5	39.7	5	5	93.40	20.2
Moutier	85.8	36.5	12	7	33.77	20.3
Neuveville	76.9	43.5	17	15	5.16	20.6
Porrentruy	76.1	35.3	9	7	90.57	26.6

上述代码使用 `head()` 函数查看了 `swiss` 数据集的前 6 行数据，它一共使用了 6 个指标来度量瑞士城市的发展程度，这 6 个指标分别是 `Fertility`（农耕地肥沃程度）、`Agriculture`（农业）、`Examination`（测验）、`Education`（教育）、`Catholic`（天主教徒）和 `Infant.Mortality`（出生率 / 死亡率）。

仅从指标名称来推断，`Fertility` 和 `Agriculture` 显然具有相关性，`Examination`、`Education` 及 `Catholic`、`Infant.Mortality` 也是两组可能具有相关关系的变量，为了检查这些变量之间是否真的具有相关性，不妨查看一下它们的相关矩阵。

```
> cor(swiss)
```

	Fertility	Agriculture	Examination	Education	Catholic
Infant.Mortality					
Fertility	1.0000000	0.35307918	-0.6458827	-0.66378886	0.4636847
Agriculture	0.3530792	1.00000000	-0.6865422	-0.63952252	0.4010951
Examination	-0.6458827	-0.68654221	1.0000000	0.69841530	-0.5727418
Education	-0.6637889	-0.63952252	0.6984153	1.00000000	-0.1538589
Catholic	0.4636847	0.40109505	-0.5727418	-0.15385892	1.0000000
Infant.Mortality	0.4165560	-0.06085861	-0.1140216	-0.09932185	0.1754959

上述代码查看了 `swiss` 数据集中各个变量组成的相关矩阵，R 返回了一个由相关系数构成的对称方阵。在方阵中最小的相关系数绝对值为 `Agriculture` 和 `Infant.Mortality` 的相关系数，其绝对值为 0.060 858 61，是一个比 0.05 更大的数，因此 `swiss` 中变量相互之间均具有或强或弱的相关关系，这份数据适合做主成分分析。

检查完数据集中变量的相关性后，还需对变量进行标准化。数据集 `swiss` 中的 `Fertility`、`Agriculture` 和 `Catholic` 具有较大的量纲，其他两个变量的量纲则较小。变量的量纲不同会使主成分得分系数的可解释性变差，此外，还会使主成分分析的结果主要受量纲较大的变量影响，而忽略量纲较小的变量。为了避免这样的问题，需要将数据进行标准化，使每个变量都服从均值为 0、方差为 1 的正态分布。

```
> std.swiss <- scale(swiss[1:6])
> head(std.swiss)
```

	Fertility	Agriculture	Examination	Education	Catholic
Infant.Mortality					
Courtelay	0.8051305	-1.4820682	-0.18668632	0.1062125	-0.7477267
	0.77503669				

```

Delemont      1.0372847  -0.2447942 -1.31480509 -0.2057867  1.0477479
0.77503669
Franches-Mnt  1.7897846  -0.4825622 -1.44015162 -0.6217858  1.2529998
0.08838778
Moutier       1.2534283  -0.6234617 -0.56272591 -0.4137863 -0.1768099
0.12272023
Neuveville    0.5409551  -0.3152440  0.06400674  0.4182118 -0.8628212
0.22571757
Porrentruy    0.4769125  -0.6762990 -0.93876550 -0.4137863  1.1851420
2.28566429

```

上述代码使用 `scale()` 函数对数据进行了标准化，并将标准化后的数据赋给了 `std.swiss`，用 `head()` 函数查看了 `std.swiss` 的前 6 行。观察 R 的返回结果，与原始数据相比，标准化后的数据大部分都位于 $-2 \sim 2$ 之间，数据标准化能够等比例地放大或缩小数据而不改变数据结构，即不影响变量间的相关关系。

9.1.2 构造一个主成分分析模型

主成分分析模型的建立需要用到相关矩阵和协方差阵等数据，R 中的基础包 `stats` 提供了用于构建主成分分析模型的函数，只需直接调用该函数即可。

```

> p <- princomp(std.swiss,cor=TRUE)
> summary(p,loadings=TRUE)
Importance of components:

               Comp.1   Comp.2   Comp.3   Comp.4   Comp.5
Comp.6
Standard deviation   1.7887865 1.0900955 0.9206573 0.66251693 0.45225403
0.34765292
Proportion of Variance 0.5332928 0.1980514 0.1412683 0.07315478 0.03408895
0.02014376
Cumulative Proportion 0.5332928 0.7313442 0.8726125 0.94576729 0.97985624
1.00000000

Loadings:
               Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6
Fertility      -0.457  0.322  0.174  0.536  0.383 -0.473
Agriculture    -0.424 -0.412         -0.643  0.375 -0.309
Examination     0.510  0.125         0.814  0.224
Education       0.454  0.179 -0.532         -0.681
Catholic        -0.350  0.146 -0.807         0.183  0.402
Infant.Mortality -0.150  0.811  0.160 -0.527 -0.105

```

上述代码使用 `princomp()` 函数构建了一个主成分分析模型，并将模型结果赋给了变量 `p`。上述代码中的 `princomp()` 函数一共设置了两个参数，第一个参数表明 `princomp()` 函数的应用对象为 `std.swiss`，`princomp()` 函数的应用对象只能为数值矩阵或数据框；第二个参数 `cor` 指定为真，表示用样本的相关矩阵做主成分分析，当该参数设定为默认值

“FALSE”时，`princomp()` 函数将使用样本的协方差矩阵做主成分分析。

`summary()` 函数查看了 p 中存储的信息，R 返回了一个列表，其中 Importance of components 元素中的 Standard Deviation 行给出了 6 个主成分的标准差；Proportion of Variance 行则给出了主成分的方差贡献率；Cumulative Proportion 行给出了方差累计贡献率。

这 6 个主成分是 6 个原始变量的投影，方差贡献率用于体现每个主成分对原始数据中信息的解释能力。第一个主成分的方差贡献率为 0.533 2，即第一个主成分能够解释原始数据中 53.32% 的信息，第二个主成分则能解释原始数据中 19.80% 的信息，越靠后的主成分能解释的信息越少。6 个主成分合起来对原始数据有百分之百的解释力，分析人员总希望用尽可能少的变量数解释尽可能多的信息，在上述例子中，前两个主成分的累计贡献率已达到 0.731 344 2，因此提取两个主成分即可获得绝大部分的原始信息。

由于 `summary()` 函数设定参数 `loadings` 为真，因此 R 同样返回了各个主成分对应的系数，根据 R 给出的系数，即列出如下两个主成分的表达式：

$$Y_2 = 0.322 \times \text{Fer} - 0.412 \times \text{Agr} + 0.125 \times \text{Exa} + 0.179 \times \text{Edu} + 0.146 \times \text{Cat} + 0.811 \times \text{Inf}$$

$$Y_1 = 0.457 \times \text{Fer} + 0.424 \times \text{Agr} + 0.510 \times \text{Exa} + 0.454 \times \text{Edu} - 0.350 \times \text{Cat} - 0.150 \times \text{Inf}$$

在上述两个公式中将变量名称简写为每个变量的前三个字母。根据主成分系数，即可将原始变量逐一投影到主成分上去。其他 4 个主成分的表达式也可以类似地写出。此外，`loadings` 给出的系数有一些是空缺值，这说明该主成分在空缺值处对应的系数特别小，不具有价值。比如第三个主成分的表达式中就没有关于 Agr 和 Exa 的系数。

值得注意的是，主成分的表达式中系数的正负本身并没有实际意义，有意义的是不同主成分之间统一变量的系数大小。比如，在第一个主成分中，前 5 个变量的系数都比较大，而 Inf 的系数则较小，这意味着第一个主成分是对前 5 个变量的一个综合测度；而第二个主成分则只有前两个变量和最后一个变量的绝对值比较大，因此第二个主成分主要体现了 Fer、Agr、Inf 中的信息。

除方差累积贡献率外，碎石图也可以帮助判断主成分分析模型中主成分的最佳个数。`screplot()` 函数提供了绘制碎石图的简便方法。

```
> screplot(p, type="lines")
```

上述代码使用 `screplot()` 函数为模型 p 绘制了碎石图，函数中设置参数 `type` 为 `lines`，表示用直线作图，当参数 `type` 为 `barplot` 时，绘制的则是直方图。R 的返回结果如图 9.1 所示， x 轴表示 6 个主成分， y 轴表示方差，观察图 9.1，从第一个主成分到第二个主成分的线段斜率较大，而在第四个主成分后直线便趋于平滑。因此，此处取 2~4 个主成分都是较为合适的。

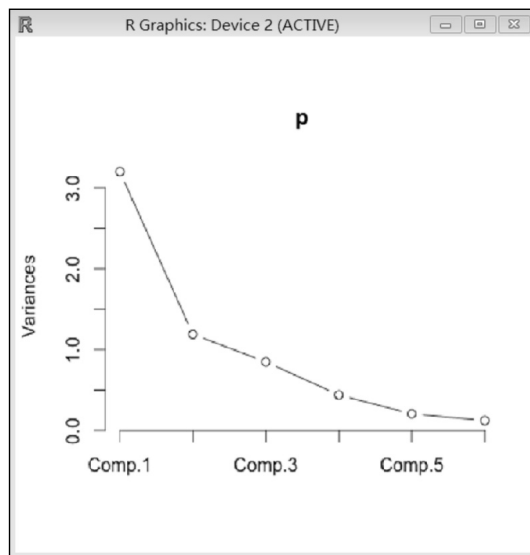


图 9.1 碎石图

9.1.3 计算主成分的综合得分

有关主成分的最后一步计算是根据主成分得分计算每一个观测值的综合得分。我们希望用一个单独的指标来衡量观测值，因此计算综合得分是一个必要的步骤。

```
> pre <- predict(p)
> head(pre)
```

	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5	Comp.6
Courtelary	0.3635516	1.3994204	0.8597075	0.9109636	-0.63161039	-0.28337042
Delemont	-1.6341755	1.0260470	-0.5478672	0.5059219	-0.64635832	-0.09123058
Franches-Mnt	-2.1042002	0.7460236	-0.4726831	1.5019113	-0.40765420	-0.08720438
Moutier	-0.7476079	0.5958917	0.5791595	1.0722987	-0.22992758	-0.30997471
Neuveville	0.3815276	0.4488448	0.6282612	0.2460774	-0.07112802	-0.76731497
Porrentruy	-1.3692190	2.2918784	-0.3505274	-0.3077827	-0.83653932	0.70933512

上述代码使用 `predict()` 函数计算了每个观测值的 6 个主成分得分，并将结果赋给了 `pre`。`head()` 函数查看了 `pre` 的前 6 行数据。R 返回了前 6 个城市在 6 个主成分上的得分。输入 `p$scores` 命令能得到同样的主成分得分矩阵。

计算主成分得分矩阵是计算综合得分的准备步骤，将主成分得分与样本数据的特征向量做内积即可得到综合得分，因此，在计算综合得分前还需计算样本数据的特征向量。

```
> y <- eigen(cor(std.swiss))
> y
$values
[1] 3.1997570 1.1883082 0.8476098 0.4389287 0.2045337 0.1208626

$vectors
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.4569876 0.3220284 0.17376638 0.53555794 0.38308893 0.47295441
[2,] 0.4242141 -0.4115132 -0.03834472 -0.64291822 0.37495215 0.30870058
[3,] -0.5097327 0.1250167 0.09123696 -0.05446158 0.81429082 -0.22401686
[4,] -0.4543119 0.1790495 -0.53239316 -0.09738818 -0.07144564 0.68081610
[5,] 0.3501111 0.1458730 -0.80680494 0.09947244 0.18317236 -0.40219666
[6,] 0.1496668 0.8111645 0.16010636 -0.52677184 -0.10453530 -0.07457754

> y1 <- y$values[1]
> y2 <- y$values[2]
> y3 <- y$values[3]
> y4 <- y$values[4]
```

上述代码使用 `eigen()` 函数计算了 `cor(std.swiss)` 的特征值和特征向量，其中，`cor(std.swiss)` 给出了 `swiss` 中数据标准化后的相关矩阵。第 2 行命令查看了 `y` 中的元素，R 返回了一个列表，第一个元素 `values` 为 `cor(std.swiss)` 的 6 个特征值；第二个元素 `vectors` 为 `cor(std.swiss)` 的 6 组特征向量。

上述代码的最后 4 行代码将前 4 个特征值分别赋给了 `y1`、`y2`、`y3` 和 `y4`。主成分分析一共给出了 6 个主成分，但我们仅使用前 4 个主成分计算综合得分，因此在此处仅取出了前 4 个特征值。

```
> scores <- (y1*pre[,1]+y2*pre[,2]+y3*pre[,3]+y4*pre[,4])/(y1+y2+y3+y4)
> head(scores)
Courtelary      Delemont Franches-Mnt      Moutier      Neuveville      Porrentruy
0.6969234      -0.7493067      -0.9847106      -0.1273211      0.4220017      -0.3682927
```

上述代码的第 1 行计算了主成分得分与相关矩阵的特征向量的内积，并与特征值的和做商，从而得到了综合得分 `scores`。查看 `scores` 中的前 6 行，R 返回了 6 个城市的综合得分。综合得分越高，城市的发展水平显然就越高，观察 R 的返回结果，`Courtelary` 是这 6 个城市中发展最好的城市，`Franches-Mnt` 则是这 6 个城市中发展最差的城市。

除计算主成分得分与相关矩阵的内积外，计算主成分得分与方差贡献率的乘积之和也是综合得分的一种计算方法。总的来说，综合得分是一个综合了全体信息的指标，具有较高的客观性和可信度，在为观测值聚类时尤其有用。

9.2 因子分析的初次构建与完善

本节将初次构建一个因子分析模型，并讨论因子分析模型中的常用参数和方法。与主成分分析相比，因子分析使用了更加复杂的降维方法。本节还将讨论因子分析与主成分分析的异同。

9.2.1 构造一个简单的因子分析模型

因子分析同样使用投影的方法将高维空间中的数据映射到低维空间中。与主成分分析不同的是，因子分析增加了一个旋转因子载荷矩阵的步骤，这一步骤进一步降低了几个因子之间的相关程度，并增加了因子的可解释性。

```
> fact <- factanal(swiss,2,scores="Bartlett",rotation="varimax")
```

R 中的基础包 `stats` 提供了用于构建因子分析模型的 `factanal()` 函数，上述代码为 `factanal()` 函数指定了 4 个参数，第一个参数指明用于构建模型的原始数据为 `swiss`，为了检验数据标准化是否会影响模型结果，在这里并未将数据进行标准化，除数据框以外，数据公式或矩阵也能用于构建因子分析模型；第二个参数指定抽取的因子个数，因子分析要求在创建模型前便指定因子个数，基于前面的分析结果，我们指定为 `swiss` 抽取两个因子。

第三个参数 `scores` 用于指定计算因子得分的方法，在上述代码中指定 `scores` 的值为

Bartlett，即使用加权最小二乘法计算因子得分，此外可供选择的参数还有表示使用回归法计算因子得分的 **regression**，**none** 则表示不计算因子得分；最后一个参数 **rotation** 指定了因子旋转的方法，默认方法为 **varimax**，即最大方差正交旋转，也可以设置为 **none**，即不旋转因子载荷矩阵。

```
> fact

Call:
factanal(x=swiss, factors = 2, scores = "Bartlett", rotation = "varimax")

Uniquenesses:
  Fertility      Agriculture      Examination      Education      Catholic      Infant.
Mortality
    0.420         0.492         0.270         0.005         0.061         0.960

Loadings:
              Factor1 Factor2
Fertility      -0.652   0.393
Agriculture    -0.631   0.333
Examination     0.685  -0.510
Education       0.997
Catholic       -0.124   0.961
Infant.Mortality      0.175

              Factor1 Factor2
SS loadings    2.311   1.481
Proportion Var  0.385   0.247
Cumulative Var  0.385   0.632

Test of the hypothesis that 2 factors are sufficient.
The chi square statistic is 20.99 on 4 degrees of freedom.
The p-value is 0.000318
```

上述代码查看了因子分析模型的具体信息。R 返回了一个列表，第一个元素 **Call** 给出了因子分析模型的摘要信息，如数据来源、因子个数、因子得分计算方法和旋转方法等；第二个元素 **Uniquenesses** 给出了 6 个原始变量的特殊方差，也就是每个变量的误差项的估计值；第三个元素 **Loadings** 给出了旋转后的载荷矩阵，由于只抽取了两个因子，因此旋转因子载荷矩阵只给出了 **Factor1** 和 **Factor2** 这两个因子，根据 **Loadings** 给出的系数，即可写出类似于主成分分析所给出的因子得分表达式。

在旋转因子载荷矩阵的下方有一个小矩阵，其中 **SS Loadings** 是公因子的方差贡献，**Proportion Var** 是每个因子的方差贡献率，**Cumulative Var** 则是累积方差贡献率。在上述模型中抽取两个因子后，累积方差贡献率达到了 63.2%，并未达到 70%，提示分析人员应该增加抽取的因子个数。

在返回结果的末尾，R 给出了有关因子分析模型可信度的信息。结果显示，关于 2 因子的假设检验是显著的，在自由度为 4 的水平上卡方检验统计量的值为 20.99，因子

分析模型的 p 值为 0.000 318, 因此, 该因子分析模型是可信的。这三行信息反映了原始数据是否适合做因子分析, 倘若原始变量不够相关, 或原始变量不服从正态分布, 那么因子分析模型将不能通过检验。

9.2.2 计算因子得分并分析

在得到因子分析的模型后, 还需进一步从模型中提炼信息, 将原始变量转化为两个因子变量, 从而得出每个城市的因子得分与综合得分, 并用于观察变量和变量之间、城市和城市之间的关系。在计算因子得分之前, 计算原始变量的列均值有利于帮助我们估计因子模型。

```
> colMeans(swiss)
      Fertility      Agriculture      Examination      Education
      70.14255      50.65957      16.48936      10.97872
      Catholic Infant.Mortality
      41.14383      19.94255
```

上述代码使用 `colMeans()` 函数对 `swiss` 数据集按列求均值。R 返回了 `swiss` 中 6 列变量的均值, 根据均值可以写出因子模型为:

$$\begin{cases} \text{Fer} - 70.14\ 255 = -0.652f_1 + 0.393f_2 \\ \text{Agr} - 50.65\ 957 = -0.631f_1 + 0.333f_2 \\ \text{Exa} - 16.48\ 936 = 0.685f_1 - 0.510f_2 \\ \text{Edu} - 10.97\ 872 = 0.997f_1 \\ \text{Cat} - 41.14\ 383 = -0.124f_1 + 0.961f_2 \\ \text{Inf} - 19.94\ 255 = 0.175f_2 \end{cases}$$

观察模型中两个因子各自的系数, 显然 `Factor1` 在变量 `Fertility`、`Agriculture`、`Examination`、`Education` 上的系数较大, 它主要反映了前 4 个变量; `Factor2` 在变量 `Examination` 和 `Catholic` 上的值比较大, 主要反映了第 3 个变量和第 5 个变量, 因此不妨为 `Factor1` 命名为农业与教育因子, 为 `Factor2` 命名为宗教教育因子。

上述结论与我们在一开始的猜想并不完全一致, 变量 `Fertility`、`Agriculture` 确实比较相似, 被映射到一个因子中, 但变量 `Catholic` 和 `Infant.Mortality` 却并未映射到一个因子中。此外, 变量 `Infant.Mortality` 并未体现在因子分析抽取出的两个因子中, 这也进一步说明我们抽取的因子个数并不十分恰当。

```
> head(fact$scores)
      Factor1      Factor2
Courtelary  0.077522661 -0.6728052
Delemont   -0.176770379  1.1433711
Franches-Mnt -0.587047649  1.2715239
Moutier     -0.426999477 -0.1693079
Neuveville  0.381945331 -0.7076872
Porrentruy -0.371323518  1.1534509
```


与主成分分析相似，因子分析模型中的 `scores` 变量存储了因子得分矩阵，上述代码查看了 `fact$scores` 的前 6 个元素，即 `swiss` 中前 6 个城市的因子得分矩阵。每个城市的 `Factor1` 值和 `Factor2` 值分别衡量了该城市在农业与教育因子和宗教教育因子方面的得分，显然，`Factor1` 值越高，城市的农业与教育就越发达，`Factor2` 值越高，城市的宗教与教育就越发达。

```
> all.scores <- fact$scores[,1]*0.385+fact$scores[,2]*0.247
> head(all.scores)
  Courtelary      Delemont Franches-Mnt      Moutier  Neuveville  Porrentruy
-0.13633665    0.21435606    0.08805307   -0.20621386   -0.02774980
0.14194281
```

上述代码使用因子得分与方差贡献率的乘积之和作为每个城市的综合得分，在第 1 行代码中为 `Factor1` 和 `Factor2` 赋的权重 0.385 和 0.247 分别是这两个因子的方差贡献率，因子的方差贡献率大，该因子就更重要，在综合得分中占的比重也就更大。

`head()` 函数查看了前 6 个城市的综合得分，此时得分最高的城市变为了 `Delemont`，而得分最低的城市则变为了 `Moutier`。与主成分分析的结果相比，城市的综合排名出现了较大的变动，结合每个因子的意义，在该模型中综合得分越高的城市在农业和教育方面也就越发达。

此外，到目前为止我们已经了解了两种综合得分的计算方式。综合得分的计算方法有许多种，在进行选择时不妨结合每个因子的实际意义及想要解决的实际问题加以考虑。

9.3 对因子分析模型进行修正

本节将在 9.2 节的基础上讨论如何修正因子分析模型。9.2 节抽取了两个因子，本节将抽取三个因子的情况与其进行对比，并使用其他方法构造不同的因子分析模型。

9.3.1 修改因子分析模型中的因子个数

在 9.2 节中通过累积方差贡献率和因子的实际意义来判定，当抽取两个因子时，累计方差贡献率较低，两个因子也不能较全面地代表原始数据中的信息。本节修改了抽取的因子个数，尝试抽取三个因子来构建因子分析模型。

```
> fact <- factanal(swiss,3,scores = "Bartlett",rotation="varimax")
> fact

Call:
factanal(x = swiss, factors = 3, scores = "Bartlett", rotation = "varimax")

Uniquenesses:
      Fertility      Agriculture      Examination      Education      Catholic
          0.005           0.286           0.213           0.114           0.083
```

```

Infant.Mortality
      0.743

Loadings:

```

	Factor1	Factor2	Factor3
Fertility	-0.512	0.203	0.832
Agriculture	-0.774	0.312	-0.129
Examination	0.751	-0.423	-0.211
Education	0.901		-0.262
Catholic	-0.186	0.913	0.220
Infant.Mortality			0.500

	Factor1	Factor2	Factor3
SS loadings	2.273	1.164	1.120
Proportion Var	0.379	0.194	0.187
Cumulative Var	0.379	0.573	0.759


```

The degrees of freedom for the model is 0 and the fit was 1e-04

```

上述代码仍使用 `factanal()` 函数构建了因子分析模型，`factanal()` 函数中的 4 个参数指定 `swiss` 数据集作为原始数据，并抽取三个因子，使用 `Bartlett` 方法计算因子得分，使用 `varimax` 方法旋转因子载荷矩阵。观察 R 返回的结果，随着抽取的因子个数发生了变化，`Uniquenesses` 也跟着发生了较大的变化，这是由于因子模型中每个原始变量的计算公式中都增加了一个新因子，因此，每个原始变量的误差项估计值也发生了变化。

`Loadings` 元素提供了因子载荷矩阵，观察该矩阵，此时第一个因子在前 4 个变量上仍具有较大的系数，但它更加侧重于反映 `Agriculture`、`Examination`、`Education` 变量，不妨为之命名为农业与教育因子；第二个因子主要反映了第五个变量，不妨为之命名为宗教因子；第三个因子则主要反映了第一个变量和第六个变量，不妨为之命名为土壤与生育率因子。为因子分析模型抽取三个因子时，仍未出现一个较好的反映第六个变量的因子，但此时比之抽取两个因子的情况已经好转了不少。另外，变量 `Fertility` 和 `Infant.Mortality` 出现了奇怪的相关关系，因子分析将这两者投射到了同一个因子上。

观察 R 返回的方差贡献率，与 9.2 节中的模型相比较，前两个因子的方差贡献率有了不同程度的减小，但总的累积方差贡献率达到了 75.9%，模型对原始数据的解释力达到了令人满意的程度。最后一行的检验信息表明该因子模型能通过检验。

```

> head(fact$scores)

```

	Factor1	Factor2	Factor3
Courtelary	0.58103001	-1.0041727	1.5754744
Delemont	-0.08158929	0.9591128	0.9687263
Franches-Mnt	-0.21325256	0.9025861	1.8029834
Moutier	-0.04878791	-0.5347301	1.6059226
Neuveville	0.43094203	-0.9408696	1.1355110
Porrentruy	-0.01001860	1.0993803	0.3241776

```

> all.scores <- fact$scores[,1]*0.379+fact$scores[,2]*0.194+fact$scores[,3]*0.187
> head(all.scores)

```

Courtellary	Delemont	Franches-Mnt	Moutier	Neuveville	Porrentruy
0.3200146	0.3362974	0.4314369	0.1780793	0.1931389	0.2701039

不妨重新查看新模型中的因子得分，上述代码中的第一行查看了前6个数据的因子得分，与9.2节中的因子得分相比，抽取三个因子后，模型中的因子得分发生了剧烈的变化，这显然是因为每个因子的因子得分计算公式发生了很大的改变，每个因子的实际含义也有所变化。此时第一个因子得分较高的城市在农业和教育方面发展较好；第二个因子得分较高的城市在宗教方面发展较好；第三个因子得分较高的城市在土壤和生育率方面较好。

根据因子得分和方差贡献率计算每个城市的总得分，此时前6个城市中得分最高的城市变为了 Franches-Mnt，得分最低的城市则没有发生变化。与抽取两个因子时相比，抽取三个因子后模型反映了更多的原始信息，因此在此模型中的城市排名应当比9.2节中的模型更加可靠。

9.3.2 基于主成分法和主轴因子法进行因子分析

在因子分析中，估计因子载荷矩阵是最重要的步骤之一，`factanal()` 函数仅提供了一种估计因子载荷矩阵的方法，即极大似然估计法。另外，两种常用的因子载荷矩阵估计方法是主成分法和主轴因子法。R 的基础包中并未提供与这两种方法相关的因子分析函数，不过程序包 `psych` 提供了相关的函数。

```
> library(psych)
> RMatrix <- cor(swiss)
> pc <- principal(r=RMatrix,nfactors=2, method="Bartlett",
rotate="varimax",scores=TRUE)
```

上述代码首先加载了程序包 `psych`，然后使用 `cor()` 函数将 `swiss` 的相关矩阵赋给了变量 `RMatrix`。第3行代码使用 `principal()` 函数构建了因子分析模型，`principal()` 函数是一种基于主成分法计算因子载荷矩阵的函数，它的参数设置与 `factanal()` 函数极为相似，上述代码为 `principal()` 函数设置了5个参数，分别指定构建模型的原始数据为 `RMatrix`；抽取两个因子；使用 `Bartlett` 方法计算因子得分；使用 `varimax` 方法旋转因子载荷矩阵；以及输出因子得分。

在这5个参数中比较特别的是 `principal()` 函数将一个相关矩阵作为应用对象，`principal()` 函数同样也允许这种用法。此外，它们还能够在协方差阵上应用函数，构建因子分析模型。

```
> pc$weight
          PC1          PC2
Fertility   0.1355176  0.366292954
Agriculture  0.3545749 -0.270232045
Examination -0.3070840  0.007356772
Education   -0.2954727  0.064645267
Catholic    0.1362968  0.194006667
Infant.Mortality -0.1828639  0.726140127
```

上述代码查看了 `pc` 中的因子载荷矩阵，将这一矩阵与9.2节中的因子载荷矩阵相对

比，显然，此二者具有较大差异。由于该模型与 9.2 节中模型的数据来源、因子个数、因子得分计算方法和因子载荷矩阵选择方法均一致，因此，两个因子载荷矩阵的差异是由因子载荷矩阵估计方法的不同引起的。

```
> fa <- fa(r=RMatrix,nfactors=2,fm="pa",method="Bartlett",rotate="varimax",
,scores=TRUE)
> fa$weights
```

	PA1	PA2
Fertility	-0.01391771	0.83921433
Agriculture	0.33628669	-0.21315863
Examination	-0.46268845	-0.04465115
Education	-0.25163497	0.28317744
Catholic	0.01022552	0.01764790
Infant.Mortality	-0.07148116	0.21107598

除 `principal()` 函数之外，程序包 `psych` 也提供了使用主轴因子法估计因子载荷矩阵的 `fa()` 函数，`fa()` 函数的参数使用格式与 `factanal()` 函数、`principal()` 函数一致，在上述代码中同样使用相关矩阵 `RMatrix` 作为原始数据；抽取两个因子；使用 `Bartlett` 方法计算因子得分；使用 `varimax` 方法旋转因子载荷矩阵；并计算因子得分。唯一一个特别的参数是 `fm` 参数指定为 `pa`，即使用主轴因子法估计因子载荷矩阵。

因子模型的因子载荷矩阵存储在 `weights` 元素中，上述代码中最后一行命令查看了其中的内容。这一因子载荷矩阵与使用主成分法估计出的因子载荷矩阵又有所不同。

```
> pcFS <- as.matrix(scale(swiss))%*%pc$weight
> head(pcFS)
```

	PC1	PC2
Courtelary	-0.63408832	1.1186300
Delemont	0.51941090	1.1891808
Franches-Mnt	0.85202806	1.0424711
Moutier	0.19732471	0.6515219
Neuveville	-0.34056961	0.3073529
Porrentruy	-0.02106062	2.2134298

```
> faFS <- as.matrix(scale(swiss))%*%fa$weight
> head(faFS)
```

	PA1	PA2
Courtelary	-0.5130010	1.1804013
Delemont	0.5186839	1.1051998
Franches-Mnt	0.6421101	1.5338732
Moutier	0.1268030	1.1155256
Neuveville	-0.2733504	0.6691610
Porrentruy	0.1531479	0.9724969

上述两条代码计算了 `swiss` 的标准化数据和因子载荷矩阵的内积作为因子得分，其中使用主成分法计算得出的因子得分赋给了变量 `pcFS`，使用主轴因子法计算得出的因子得分赋给了变量 `faFS`。分别查看这两种因子得分的前 6 行数据，并与 9.2 节中使用极大似然法估计出的因子得分相比较，显然，每种方法的因子得分都不相似。

9.4 在降维分析的基础上进行回归分析和聚类分析

本节的主题是本章的最后一部分内容，即如何将降维分析与其他分析方法结合起来。降维分析将多个互相之间存在相关关系的变量减少为少数互相之间不存在相关关系的变量，本节关心的是如何将降维分析的结果应用在其他分析方法中。

9.4.1 在降维分析的基础上进行回归分析

在本章的开头曾提到过，降维分析原本就是为了满足回归分析的需求才发展起来的。降维分析能非常自然地解决回归分析中的多重共线性问题。

```
> p <- princomp(scale(swiss),cor=TRUE)
> x <- as.data.frame(p$scores)
> head(x)
```

	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5	Comp.6
Courtellary	0.3635516	1.3994204	0.8597075	0.9109636	-0.63161039	-0.28337042
Delemont	-1.6341755	1.0260470	-0.5478672	0.5059219	-0.64635832	-0.09123058
Franches-Mnt	-2.1042002	0.7460236	-0.4726831	1.5019113	-0.40765420	-0.08720438
Moutier	-0.7476079	0.5958917	0.5791595	1.0722987	-0.22992758	-0.30997471
Neuveville	0.3815276	0.4488448	0.6282612	0.2460774	-0.07112802	-0.76731497
Porrentruy	-1.3692190	2.2918784	-0.3505274	-0.3077827	-0.83653932	0.70933512

上述代码中的第一行代码构建了一个主成分分析模型，并将模型结果存储在变量 p 中；第2行代码则将 p 中的 `scores` 变量转换为数据框后存储在变量 x 中。在本章的9.1节中已经提到了有关如何建立主成分分析模型的内容，也介绍了有关主成分得分的知识。此时 x 中存放的即为47个瑞士城市的6个主成分得分。这6个主成分得分是由6个原始变量转换而来的，不妨使用前两个主成分得分作为自变量，来代表原始数据。

```
> y <- rowMeans(scale(swiss))
> head(y)
```

	Courtellary	Delemont	Franches-Mnt	Moutier	Neuveville	Porrentruy
	-0.12168358	0.18244722	0.09777877	-0.06677254	0.01180433	0.31981134

上述代码为回归分析模型构造了一个因变量。在 `swiss` 数据集中并未提供合适的自变量，因此，上述代码求得了 `swiss` 数据集中6个原始变量标准化后的均值。在第1行代码中 `scale()` 函数用于标准化数据，`rowMeans()` 函数则用于对数据集按行求均值。查看 y 中的前6个元素，它为每一个城市存储了6个原始变量的均值。

我们打算将变量 y 作为因变量纳入回归模型中，仅从变量的实际意义上考虑，变量 y 与变量 x 都和数据集 `swiss` 中的数据有直接联系，因此，这二者之间的回归关系应当是显著的。不妨实际地查看一下最终结果。

```
> lm1 <- lm(y~Comp.1+Comp.2,x)
> summary(lm1)
```

Call:

```
lm(formula = y ~ Comp.1 + Comp.2, data = x)

Residuals:
    Min       1Q   Median       3Q      Max
-0.36703 -0.17831  0.01597  0.15245  0.49862

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.534e-17  3.101e-02   0.000 1.000000
Comp.1       -6.875e-02  1.733e-02  -3.966 0.000266 ***
Comp.2        1.932e-01  2.845e-02   6.791 2.31e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2126 on 44 degrees of freedom
Multiple R-squared:  0.5843,    Adjusted R-squared:  0.5654
F-statistic: 30.92 on 2 and 44 DF,  p-value: 4.101e-09
```

上述代码使用 `lm()` 函数构建了回归模型，`lm()` 函数中指定因变量为 `y`，自变量为 `Comp.1` 和 `Comp.2`，`Comp.1` 和 `Comp.2` 都取自 `x`，即为主成分分析为 `swiss` 数据集计算出的前两个主成分得分。使用 `summary()` 函数查看回归模型的详细信息，该回归模型的总的 p 值为 $4.101e-09$ ，回归模型的整体是显著的。但回归模型的 R 方仅有 58.43%，并不能令人满意。

观察回归模型的 `Coefficients` 元素，`Comp.1` 和 `Comp.2` 的 p 值都比较小，是可信的，但常数项的 p 值为 1，因此我们考虑从模型中删去该项。

```
> lm1 <- lm(y~Comp.1+Comp.2-1,x)
> summary(lm1)

Call:
lm(formula = y ~ Comp.1 + Comp.2 - 1, data = x)

Residuals:
    Min       1Q   Median       3Q      Max
-0.36703 -0.17831  0.01597  0.15245  0.49862

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
Comp.1 -0.06875      0.01714  -4.011 0.000226 ***
Comp.2  0.19318      0.02813   6.868 1.6e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2102 on 45 degrees of freedom
Multiple R-squared:  0.5843,    Adjusted R-squared:  0.5658
F-statistic: 31.63 on 2 and 45 DF,  p-value: 2.644e-09
```

上述代码构建了不含常数项的回归模型。在第 1 行代码中自变量和因变量的设置都

没有变，但在设置自变量参数时添加了一项“-1”，这表示从回归方程中删去常数。仍旧使用 `summary()` 函数查看回归模型的详细信息。

在回归模型中删去常数项后，回归模型总的 p 值略有下降，回归模型的可信度上升了一些。而回归模型的 R 方并未发生变化，仍处于一个较低的水平。这可能是因为使用的主成分得分较少，仅能代表原始数据中的部分信息；也可能是因为降维的方法选择得并不恰当。

```
> f <- factanal(swiss,3,scores = "regression",rotation="varimax")
> x <- as.data.frame(p$scores)
> y <- rowMeans(swiss)
```

与主成分分析类似，上述代码使用 `factanal()` 函数构建了因子分析模型，并将模型结果赋给了变量 f 。变量 x 中存放了 47 座城市的因子得分，变量 y 则存放了 `swiss` 数据集中每座城市的 6 个原始数据的均值，根据 9.2 节的结论，不妨使用 x 中前三个因子得分作为自变量，使用 y 作为因变量构建回归模型。

```
> lm2 <- lm(y~ Factor1+Factor2+Factor3,x)
> summary(lm2)

Call:
lm(formula = y ~ Factor1 + Factor2 + Factor3, data = x)

Residuals:
    Min       1Q   Median       3Q      Max
-3.859 -1.657 -0.311  1.196  3.940

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   34.8928     0.3078 113.363 < 2e-16 ***
Factor1        -2.8590     0.3279  -8.720 4.65e-11 ***
Factor2         8.2343     0.3266  25.215 < 2e-16 ***
Factor3         2.1426     0.3191   6.715 3.33e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.11 on 43 degrees of freedom
Multiple R-squared:  0.9476,    Adjusted R-squared:  0.9439
F-statistic: 259.2 on 3 and 43 DF,  p-value: < 2.2e-16
```

上述代码使用 y 作为因变量，`Factor1`、`Factor2`、`Factor3` 作为自变量构建了回归模型，`Factor1`、`Factor2`、`Factor3` 是 x 中存放的前三个因子得分。查看回归模型的结果，模型的整体 p 值小于 0.05，总的来看，回归模型是可信的。回归模型的三个自变量和常数的 p 值都小于 0.001，都是非常可信的，因此无须对模型进行调整。

与 `lm1` 不同，`lm2` 的 R 方达到了 94.76%，它能解释原始数据中 94.76% 的信息，显然，`lm2` 要比 `lm1` 更好。利用因子分析，只需用三个变量即可表达 6 个原始变量中的绝大部分信息，这正体现了降维分析的方便之处。

9.4.2 在降维分析的基础上进行聚类分析

聚类分析是另一类能与降维分析完美地结合在一起的分析方法。以 `swiss` 数据集为例，聚类分析能在降维分析的基础上进一步探究 47 所瑞士城市的相似程度，并找出其中的规律。除去为城市聚类外，聚类分析同样也能和降维分析相结合，为企业品牌、个人、旅游景点等评分并聚类。

```
> s1 <- p$scores[,1]*0.5332928+p$scores[,2]*0.1980514
> head(s1)
  Courtelary      Delemont Franches-Mnt      Moutier  Neuveville  Porrentruy
0.4710366   -0.6682840   -0.9744038   -0.2806767   0.2923603   -0.2762849
> disma <- dist(s1,method = "euclidean")
> clur <- hclust(d=disma,method="ward.D")
> plot(clur,main="Princomp.Cluster")
```

上述代码首先计算了 `swiss` 数据集在主成分分析下的综合得分，其综合得分计算公式中 `p$scores[,1]` 与 `p$scores[,2]` 为 `swiss` 数据集的前两个主成分得分，0.5 332 928 与 0.1 980 514 则为这两个主成分的方差贡献率，由于在主成分分析中前两个主成分的累计方差贡献率已不低于 70%，因此仅使用前两个主成分来计算综合得分。

查看 `s1` 中的前 6 个元素，`s1` 中仅存储了一个变量，这一个指标便可以代表 6 个原始变量中的大部分信息。使用 `s1` 为 `swiss` 做聚类分析，其结果与使用主成分得分做聚类分析的结果并无不同，而与直接使用原始变量相比，使用 `s1` 可以消除原始变量之间相关性对聚类结果的影响。

`dist()` 函数计算了 `s1` 的距离矩阵，利用 `hclust()` 函数根据距离矩阵构建了系统聚类模型，而利用 `plot()` 函数则绘出了树状图，这三个函数在第 8 章有关聚类分析的内容中有详细的介绍。

图 9.2 是 R 返回的系统聚类树状图。观察图 9.2, 47 座瑞士城市较为明显地分为三类，其中居于树状图左端和右端的类别中城市数目较多，居于中间的城市数据则较少。在解释聚类结果时依据主成分得分进行解释比较方便，从原始变量中则不易直接看出聚类依据。此外，结合实际意义进行解释也是非常必要的。

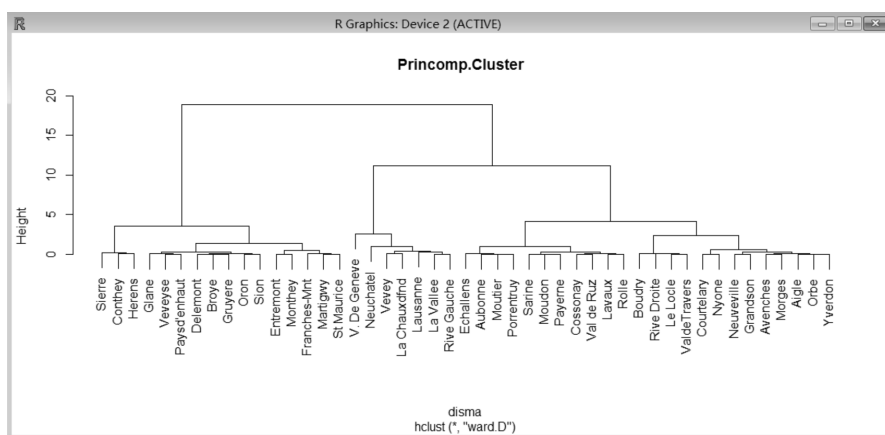


图 9.2 利用主成分分析 `swiss` 聚类


```

> s2 <- f$scores[,1]*0.379+f$scores[,2]*0.194+f$scores[,3]*0.187
> disma <- dist(s2,method = "euclidean")
> clur <- hclust(d=disma,method="ward.D")
> plot(clur,main="Factanal.Cluster")

```

上述代码同样使用因子得分与方差贡献率的乘积作为综合得分的计算公式。s2 使用了三个因子得分来计算综合得分，最终绘出的树状图如图 9.3 所示。

图 9.3 主要将瑞士城市聚为两类，对比图 9.3 与图 9.2，图 9.2 中居于中间和居于右端的两类主要合并为图 9.3 中居于左端的类，但也有一些在图 9.2 中属于一类的城市被拆开，实际上，不同的降维方法、不同的因子载荷矩阵估计方法、不同的综合得分计算方法都会引起聚类结果的变化。与主成分分析相似，图 9.3 的聚类结果同样可以从因子得分中找到依据。

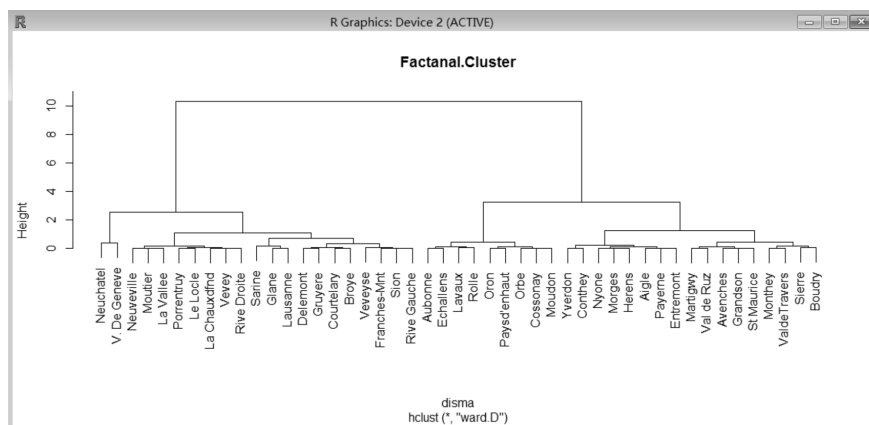


图 9.3 利用因子分析为 swiss 聚类

第 10 章 R 中的广义线性回归模型

本章的主题是广义线性回归模型，作为一般线性回归模型的补充，本章关心的是当样本数据不满足线性关系时的解决办法。本章将主要介绍一般的广义线性回归模型、logistics 回归模型和泊松回归模型，并讨论有关交叉验证的内容。

10.1 一般的广义线性回归模型

本节将讨论广义线性回归模型的意义，并介绍最简单的广义线性回归模型。通过对广义线性回归模型与一般线性回归模型的对比，本节还将讨论广义线性回归模型的适用范围。

10.1.1 使用二次函数拟合线性回归模型

线性回归模型的通式为 $Y=a_0+a_1X_1+a_2X_2+\cdots$ ，虽然在第 6 章中没有明确指出，但一般的相关分析和回归分析研究的都是变量之间具备线性关系的问题。但自变量和因变量之间的依赖关系显然不仅有线性关系，比如当 Y 与 X 满足 $Y=e^X$ 关系时， Y 与 X 显然是相互依赖的，但它们并不满足线性关系，此时相关分析和回归分析对它们也不起作用。

显然，一般线性模型是广义线性模型的一种特例，而广义线性模型将自变量和因变量之间的依赖关系推广到更复杂的情况。不妨仍以第 6 章中出现的 iris 数据集为例，并尝试为它们构建广义线性回归模型。

```
> attach(iris)
> plot(Sepal.Length~Petal.Length)
```

上述代码绑定了 iris 数据集，并使用 plot() 函数绘出了以 Petal.Length 为横轴，以 Sepal.Length 为纵轴的散点图。R 返回的结果如图 10.1 所示。观察图 10.1，图中的散点基本呈带状从图形的左下角斜跨右上角，但这种带状分布并不是非常均匀的带状分布，大部分散点分散在右上角，还有一些点较集中地处于左下角。

我们既可以让一条直线穿过这些散点并处于散点中央位置，也可以让一条曲线穿过这些散点并处于中央位置。二次曲线是最简单的曲线，不妨在散点图上添加一条二次曲线。

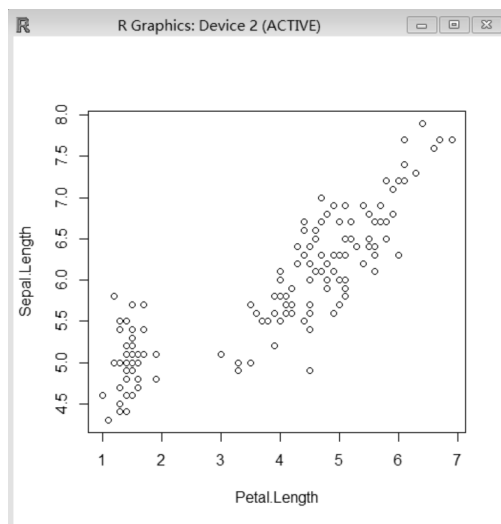


图 10.1 iris 中样本数据散点图

```
> x <- seq(1,7,0.1)
> y <- 0.2*(x-2.5)^2+5
> points(x,y,type="l")
```

上述代码首先创建了一个从 1 到 7，间隔为 0.1 的序列并赋给变量 x ，然后创建了变量 y ， y 与 x 满足关系式 $y = 0.2 \times (x - 2.5)^2 + 5$ ，显然 x 和 y 都是长度为 61 的数字序列，且 y 是关于 x 的二次函数。上述代码的第三行命令使用 `points()` 函数在图 10.1 的基础上添加了一些点，这些点以 x 为横坐标，以 y 为纵坐标，并设定参数 `type` 为 `l`，即绘制线条。

图 10.2 是 R 的返回结果。这是一条根据散点分布估计出的曲线，它基本上很好地穿过了全部散点，此时似乎使用上述代码给出的二次函数也能较好地拟合方程。为了确定我们的猜想，不妨实际检验一下该二次函数是否能通过检验。

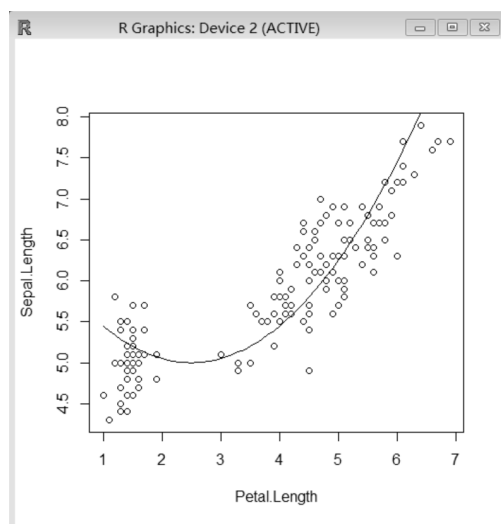


图 10.2 在散点图上添加二次曲线

```

> Petal.Length1 <- 0.2*(Petal.Length-2.5)^2+5
> lm <- lm(Sepal.Length~Petal.Length1)
> summary(lm)

Call:
lm(formula = Sepal.Length ~ Petal.Length1)

Residuals:
    Min       1Q   Median       3Q      Max
-1.08964 -0.28546 -0.02757  0.26602  1.12964

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.88951     0.24798   3.587 0.000454 ***
Petal.Length1 0.83459     0.04136  20.179 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.429 on 148 degrees of freedom
Multiple R-squared:  0.7334,    Adjusted R-squared:  0.7316
F-statistic: 407.2 on 1 and 148 DF,  p-value: < 2.2e-16

```

上述代码首先计算了当 x 为 `Petal.Length` 时二次函数 $y = 0.2 \times (x - 2.5)^2 + 5$ 中 y 的值，并将 y 的值赋给了 `Petal.Length1`，由于 `lm()` 函数并不能直接输入方程式，因此需要事先计算一个二次函数表达式的值。第 2、3 行代码使用 `lm()` 函数拟合了线性回归方程，并使用 `summary()` 函数查看了回归模型中的具体信息。

`R` 返回了方程中的系数，根据系数可知方程为 $\text{Sepal.Length} = 0.88951 + 0.83459 \times \text{Petal.Length1}$ ，使用 `Petal.Length` 替换 `Petal.Length1` 后，有 $\text{Sepal.Length} = 0.166915 \times (\text{Petal.Length} - 2.5)^2 + 5.06246$ 。这显然是一个二次函数形式的回归方程。在这个模型中，方程的 p 值与每个系数的 p 值都小于 0.005，而方程的 R 方达到 73.34%，这显然是一个还不错的模型。

10.1.2 拟合更多的广义线性模型

广义线性模型是一个大家族，仅以二次函数来说，通过修改函数的系数，即可得到无穷多个二次函数。修改自变量的次数也能得到其他新的广义线性模型。观察变量的散点图并找出合适的拟合方程固然是可行的，但这需要耗费较多时间，尝试构建基本的函数模型在构建模型时具有较高的指导作用。

```

> Petal.Length2 <- Petal.Length^2
> lm1 <- lm(Sepal.Length~Petal.Length2)
> summary(lm1)

Call:
lm(formula = Sepal.Length ~ Petal.Length2)

```

```

Residuals:
      Min       1Q   Median       3Q      Max
-1.12179 -0.27077  0.01064  0.22795  0.88533

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   4.829913   0.050697   95.27  <2e-16 ***
Petal.Length2 0.058858   0.002377   24.76  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3664 on 148 degrees of freedom
Multiple R-squared:  0.8055,    Adjusted R-squared:  0.8042
F-statistic: 613.1 on 1 and 148 DF,  p-value: < 2.2e-16

```

上述代码首先计算了 `Petal.Length` 的平方，并赋给了变量 `Petal.Length2`。第 2、3 行命令使用 `lm()` 函数为变量 `Sepal.Length` 和 `Petal.Length2` 构建线性回归模型 `lm1`，并使用 `summary()` 函数查看了模型中的具体信息。

根据 R 返回的结果，此时有拟合方程 $\text{Sepal.Length} = 0.058\ 858 \times \text{Petal.Length}^2 + 4.829\ 913$ ，与 `lm` 相比，`lm1` 具有更简洁的方程形式，此时模型的 p 值与每个系数的 p 值都小于 0.001，方程是非常可信的。此外 `lm1` 的 R 方达到 80.55%，比 `lm` 更高，说明 `lm1` 是更佳的模型。

与构建二次模型类似，R 同样也能构建三次模型、四次模型。同一个回归方程中也能引入多个高次项，比如 `lm` 模型中就既有一次项也有二次项。而包含多个次项的线性回归方程同样可以归纳出通式 $Y = a_0 + a_1X + a_2X^2 + a_3X^3 + \cdots + a_pX^p$ ，其中仅含一个自变量 X ，因此该通式又称为一元多项式回归模型，当多项式回归模型中的自变量多于一个时，模型中还可引入多个自变量的交叉乘积形式。

```

> lm2 <- lm(Sepal.Length~exp(Petal.Length))
> summary(lm2)

Call:
lm(formula = Sepal.Length ~ exp(Petal.Length))

Residuals:
      Min       1Q   Median       3Q      Max
-1.50720 -0.33166  0.02031  0.32021  1.22901

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.3428054   0.0502286  106.37  <2e-16 ***
exp(Petal.Length) 0.0038945   0.0002349   16.58  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4916 on 148 degrees of freedom

```

```
Multiple R-squared: 0.6499, Adjusted R-squared: 0.6476
F-statistic: 274.8 on 1 and 148 DF, p-value: < 2.2e-16
```

除多项式回归模型外，指数形式是另一类常见的广义线性模型。只涉及一个自变量的一元指数函数的通式为 $Y = \exp(a_1X + a_2X^2 + \dots)$ ，与多项式相似，指数函数同样可以拓展到多元的情况，并在指数部分引入交叉积。

上述代码拟合了最简单的指数回归模型，根据 R 的返回结果，此时 `lm()` 函数为自变量和因变量拟合的回归方程为 $\text{Sepal.Length} = 0.003\ 894\ 5 \times \exp(\text{Petal.Length}) + 5.342\ 805\ 4$ ，方程中指数部分的系数比较小，因此这是一个较平坦的、弯曲程度较不明显的曲线。方程 `lm2` 的 p 值及每个参数的 p 值都小于 0.001，因此，这是一个相当可信的方程。但 `lm2` 的 R 方仅为 64.99%，并不如 `lm1` 或 `lm` 好。

10.1.3 比较线性模型的优劣

一般线性回归模型是比较多项式回归模型、指数回归模型优劣的基准。如下代码使用 `lm()` 函数构建了最简单的线性回归模型。

```
> lm3 <- lm(Sepal.Length~Petal.Length)
> summary(lm3)

Call:
lm(formula = Sepal.Length ~ Petal.Length)

Residuals:
    Min       1Q   Median       3Q      Max
-1.24675 -0.29657 -0.01515  0.27676  1.00269

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  4.30660    0.07839   54.94  <2e-16 ***
Petal.Length  0.40892    0.01889   21.65  <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4071 on 148 degrees of freedom
Multiple R-squared:  0.76, Adjusted R-squared:  0.7583
F-statistic: 468.6 on 1 and 148 DF, p-value: < 2.2e-16
```

上述代码使用 `summary()` 函数查看了一元线性回归模型中的具体信息。根据 R 返回的结果，此时有回归方程 $\text{Sepal.Length} = 0.408\ 92 \times \text{Petal.Length} + 4.306\ 60$ 成立，与 `lm`、`lm1`、`lm2` 相比，这 4 个回归方程的截距项都大致相同，含自变量的项的系数则随自变量的次数增加而减小。模型 `lm3` 的 p 值与各参数的 p 值都小于 0.001，方程整体是可信的。

既然多项式线性模型、指数线性模型与广义线性模型都是可信的，则这些模型拟合

出的回归曲线应当较为相似。不妨在散点图上添加这些回归曲线，直观地看一看它们有没有优劣之分。

```
> plot(Sepal.Length~Petal.Length,xlim=c(0,10),ylim=c(0,10))
> abline(lm1)
> abline(lm2,col="red")
> abline(lm3,col="blue")
```

上述代码使用 `plot()` 函数绘出了以 `Petal.Length` 为横轴，以 `Sepal.Length` 为纵轴的散点图，并用参数 `xlim` 和参数 `ylim` 指定散点图的横轴范围和纵轴范围均为从 0 到 10。扩大坐标轴的范围有助于分析人员更清楚地看出不同回归曲线的走向及其差异。接下来的三条 `abline()` 语句分别在散点图上添加了黑色的多项式回归曲线、红色的指数回归曲线及蓝色的一般线性回归曲线，最终结果如图 10.3 所示。

观察图 10.3，蓝色曲线明显地向右上角倾斜，贯穿全体散点，并很好地反映出散点的散布趋势。黑色曲线稍次之，也表现出较弱的倾斜趋势，但它的位置偏离了中心位置，居于全体散点中心位置的较下方。红色曲线则是一条平平的曲线，位置较靠右的大部分散点都在红色曲线和黑色曲线的上方。仅观察图 10.3，一般的线性回归模型显然是最好的拟合模型。

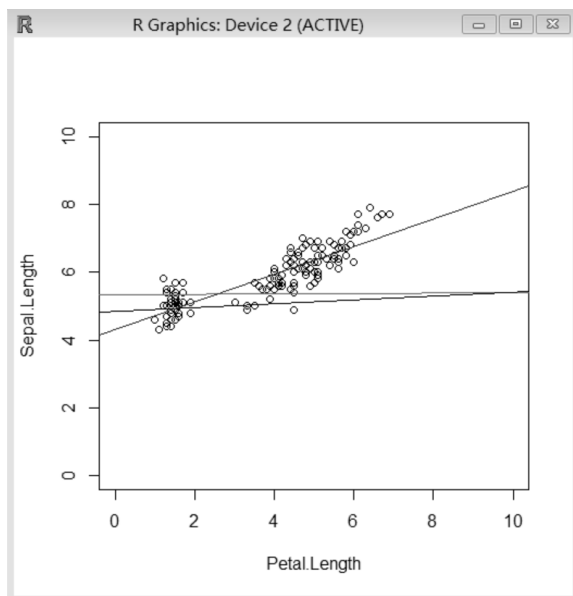


图 10.3 在散点图上添加不同的回归曲线

回想 `lm1`、`lm2` 和 `lm3` 的 R 方，它们分别为 80.55%、64.99% 和 76%，多项式回归模型的 R 方比一元线性回归模型的更高，即多项式回归模型能解释更多的原始信息，这一结论与图 10.3 并不相符。一种合理的解释是 `iris` 中的异常值影响了模型 R 方。根据图 10.3，我们仍认为一般的一元线性回归模型表现更好。

有关广义线性模型另一点值得指出的是，在多项式回归模型和指数模型中尽管自变量和因变量并不成线性关系，但自变量的系数和因变量仍满足线性关系，因此它们仍

属于线性模型。还有一些函数属于非线性模型，比如函数 $Y = \frac{a_1}{a_1 + a_2} \times (\exp(-a_3 X) + \exp(-a_4 X^2))$ 中自变量的参数和因变量不成线性关系，因此，依据该函数构建的回归模型就是非线性模型。广义线性模型与非线性模型的另一重要区别是广义线性模型能够转化为一般的线性模型，比如对指数函数取对数等，而非线性模型则无法转化。

比较广义线性模型和一般线性模型的优劣，显然广义线性模型并不比一般线性模型具有更多的优势，由于一般的线性模型计算方便，且具有较好的普适性，因此仅在数据呈现非常明显的非线性分布时才考虑使用广义线性模型或非线性模型。

10.2 Logistic 线性回归模型

本节将讨论当因变量是分类变量时如何构建模型，并讨论 Logistic 模型的显著性检验和优势比。同时还将介绍 Logistic 模型中常见的警告信息，以及这些警告信息的处理方法。

10.2.1 Logistic 模型的原理与构建方法

一般的线性模型总要求因变量是连续变量，但在现实生活中我们研究的问题中往往会出现因变量是分类变量的情况。比如研究顾客对产品的满意程度、病人是否能够康复、植物是否能够存活等。当因变量是分类变量时严重不符合线性模型的假设，此时需要使用 Logistic 模型构建回归模型。

Logistic 模型关心的是当自变量的值已知时，因变量属于某种类别的概率。二值 Logistic 模型是最基础的 Logistic 模型，它认为因变量只有两个可能取到的值，即 0 和 1。Logistic 模型使用概率推测因变量所属类别，图 10.4 所示为二值 Logistic 模型判断因变量类别的示意图。

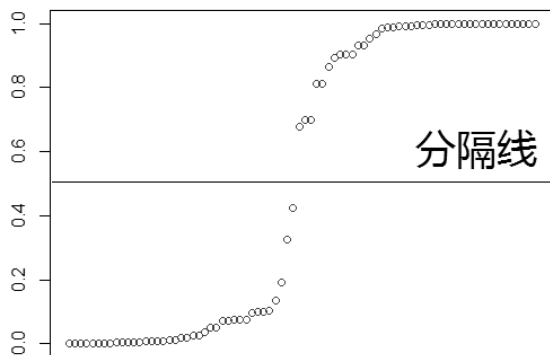


图 10.4 二值 Logistic 模型判断因变量类别的示意图

二值 Logistic 模型用于计算因变量类别为 1 的概率，图 10.4 是一个典型的 Logistic 分布模型，它标出了每个样本的概率值，其中标出了一条概率为 0.5 的分隔线，显然，居于分隔线下方的样本点都是第 0 类，居于分隔线上方的样本点都是第 1 类。

图中样本点呈明显的 S 形，落于左端的样本点的因变量类别为 1 的概率接近 0，它们的类别显然为 0；落于右端的样本点的因变量类别为 1 的概率接近 1，它们的类别显然为 1，这些点对应远离分割面的点（在二维空间中，分割两个类别的是线，三维空间中，分割两个类别的是面）。而居于中间的样本点则立场不那么鲜明，它们对应的是位于分割面附近、较难确定类别的点。

R 提供了 `glm()` 函数用于构建 Logistic 回归模型，在尝试对 `iris` 数据集中花朵的类型进行预测之前，还需对数据进行预处理。

```
> iris[,6] <- c(rep(0,50),rep(1,100))
> iris$V6 <- as.factor(iris$V6)
```

利用 `glm()` 函数仅能处理二值 Logistic 回归问题，因此，首先要构建一个二值变量。上述代码为 `iris` 添加了一系列变量，该变量的前 50 个值均为 0，后 100 个值均为 1，显然，0 对应的是花朵类型为 `setosa` 的样本，1 对应的是花朵类型不为 `setosa` 的样本，由于这列变量是 `iris` 的第六列，因此 R 自动为其命名为 `V6`。

由于 `V6` 中的 0 和 1 并不具有数值意义，仅表示类别的区分，因此，上述代码的第二行命令使用 `as.factor()` 函数将 `V6` 转换成因子型，Logistic 模型仅接受因子型的因变量。

```
> library(foreign)
> lm.log <- glm(V6~Sepal.Length,family=binomial,data=iris)
> summary(lm.log)

Call:
glm(formula = V6 ~ Sepal.Length, family = binomial, data = iris)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.14316  -0.31399   0.04605   0.27914   2.25787

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -27.8285     4.8276  -5.765 8.19e-09 ***
Sepal.Length   5.1757     0.8934   5.793 6.90e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 190.954  on 149  degrees of freedom
Residual deviance:  71.836  on 148  degrees of freedom
AIC: 75.836

Number of Fisher Scoring iterations: 7
```

上述代码首先加载了 `foreign` 程序包，第 2 行代码使用 `glm()` 函数创建了 Logistic 模型 `lm.log`。`glm()` 函数的使用格式与 `lm()` 函数相似，函数中第一个参数用于指定自变量

和因变量，上述代码中指定 Logistic 模型的自变量为 Sepal.Length，因变量为 V6。参数 family 用于指定构建模型的分布函数，上述代码中指定参数 family 为 binomial，即认为因变量服从二项分布。参数 data 则用于指定数据来源。

summary() 函数查看了模型 lm.log 的具体信息。Deviance Residuals 元素同样给出了模型的残差，观察 R 返回的结果，lm.log 的残差大致位于 -2~2 之间，样本数据中并未出现异常值。

Coefficients 元素则给出了模型的具体系数，将 R 返回的系数代入 Logistic 回归方程，可知模型 lm.log 对应的 Logistic 回归方程即为 $p(V6=1|Sepal.Length) = \frac{\exp(-27.8285 + 5.1757 \times Sepal.Length)}{1 + \exp(-27.8285 + 5.1757 \times Sepal.Length)}$ ，该方程计算了当 Sepal.Length 已知时，V6 为 1 的概率，即样本不属于 setosa 类的概率，该值显然是一个大于 0 小于 1 的值。当方程的结果大于 0.5 时，样本就不属于 setosa 类；当方程的结果小于 0.5 时，样本就属于 setosa 类。

在 R 的返回结果中，常数项和自变量 Sepal.Length 的 p 值均小于 0.001，因此方程是显著可信的，但仅有自变量系数的 p 值还不能完全确认方程的显著性，还需对方程进行卡方检验。

10.2.2 Logistic 模型的显著性检验和优势比

由于因变量服从二项分布，因此对模型的方差检验需使用似然比卡方检验。anova() 函数提供了检验 Logistic 模型的似然卡方比的功能，它也能逐一地检验模型的自变量是否显著。

```
> anova(lm.log, test="Chisq")
Analysis of Deviance Table

Model: binomial, link: logit

Response: V6

Terms added sequentially (first to last)

              Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
NULL                      149      190.954
Sepal.Length  1      119.12      148      71.836 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

上述代码使用 anova() 函数对模型 lm.log 进行了似然比卡方检验。返回结果的第 5~7 行给出了一张表格，其中 Deviance 给出了自变量的模型偏差，也就是似然比卡方。Sepal.Length 能够产生 119.12 的模型偏差，其 p 值小于 0.001，对模型的影响是十分显著的。

模型偏差度量了自变量对模型的影响程度，该值越小，就说明自变量造成的偏差越小，对因变量的分类越没有帮助；该值越大，就说明自变量造成的偏差越大，对因变量的分类帮助越多。根据模型偏差的显著与否，能较合理地判断自变量是否适合引入 Logistic 回归模型。

似然卡方比检验的结果应与 Logistic 模型的结果结合起来筛选自变量。似然卡方比检验独立计算每一自变量的模型偏差，Logistic 模型则考虑了自变量之间的交互作用。在自变量多于一个时，有些自变量的模型偏差能通过检验，但 Logistic 模型中该自变量则通不过检验，这是由于自变量之间的相关关系影响了检验结果。此时，应综合考虑两种检验的结果，多次筛选自变量，直至两种检验都通过为止。

```
> exp(coef(lm.log))
(Intercept) Sepal.Length
8.207799e-13 1.769201e+02
```

与线性模型不同，Logistic 回归模型的通式不是线性函数，其自变量的参数解释起来也较为复杂。我们不能直观地解释 Coefficients 元素给出的自变量系数，为了直接解释，还需对自变量系数取指数，从而得到每一自变量的优势比。

上述代码查看了 coef(lm.log) 的指数，coef(lm.log) 给出了常数项和 Sepal.Length 的系数，即 -27.828 5 和 5.175 7，exp() 函数查看了此二者的指数，即优势比。常数的优势比并不具有意义，Sepal.Length 的优势比则表示每当 Sepal.Length 增加一个单位，样本属于类别 1 的概率就增加 17.692 01 倍。考虑到变量 Sepal.Length 以 0.01 作为变化单位，优势比给出的概率增长速度处于一个合理的范围。

当 Logistic 模型中自变量多于一个时，优势比能够拿来相互比较，这可以用于为自变量的重要性排序，或帮助删去那些不能帮助因变量分类的自变量，即优势比无限接近 1 的自变量。

10.2.3 修正被警告的 Logistic 模型

与一般线性模型相比，Logistic 模型除要求自变量是分类变量外，其余并无不同。但 Logistic 模型要求样本呈与图 10.4 相似的 S 形分布，否则 R 将会报错。

```
> lm11.log <- glm(V6~Sepal.Length+Sepal.Width,family=binomial,data=iris)
警告信息:
1: glm.fit:算法没有聚合
2: glm.fit:拟合概率算出来是数值0或1
```

上述代码以 V6 为因变量，以 Sepal.Length 和 Sepal.Width 为自变量构建了 Logistic 模型 lm11.log，与 lm.log 相比，lm11.log 仅添加了一个自变量，但模型给出了两个警告。

第一个警告信息为算法没有聚合，即算法没有收敛。Logistic 模型依照极大似然估计原则迭代的计算方程的解，其迭代次数默认为 25 次，在默认迭代次数中未能使算法收敛，这说明样本数据不太好，数据较分散或不完全满足模型假设等，此时可以通过增加迭代次数使其收敛，但这会牺牲模型的效率。

第二个警告信息为拟合概率算出来是 0 或 1。联系图 10.4，这意味着模型中仅出现了居于端点的样本点，居于中央位置的样本点没有出现。

当 Logistic 模型出现第二条警告信息时，即意味着此时样本数据呈现如图 10.5 所示的分布。观察图 10.5，此时所有的数据处于两个极端，与图 10.4 对比，此时样本数据中那些较难分类的点已消失不见，所有的点都有一个明确的类别，这违反了 Logistic 模型的假设，因此模型会报错。

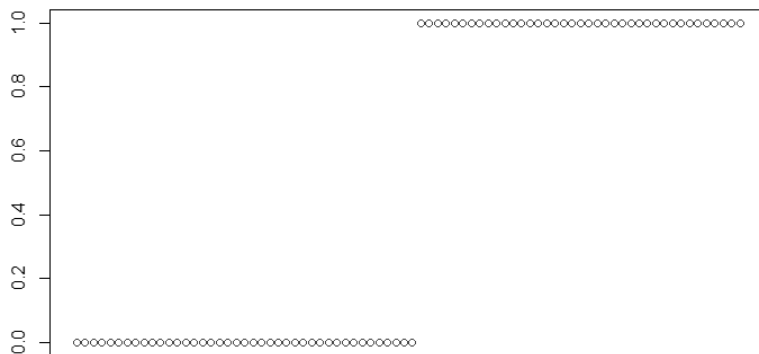


图 10.5 不呈 S 曲线的样本数据

显然，此时使用判别分析或决策树即可解决分类问题，Logistic 模型反而不能发挥应有的作用。在第 7 章中绘制分布图时我们已经发现，setosa 类别与其他两类的区别最明显，它具有最低的均值和最集中的分布，因此，使用几个较简单的条件即可判别 setosa 的模型，setosa 类别与其他两类的明显区别意味着 lm11.log 中的样本数据并不呈 S 曲线，而是呈如图 10.5 所示的分布。

如果将自变量中类别为 virginica 的样本值记为 0，将类别不为 virginica 的样本值记为 1，则上述代码将不会报出第二条警告信息，这是由于 virginica 与其他类别的区分较不明显，样本数据呈 S 形分布的原因。

```
> lm12.log <- glm(V6~Sepal.Length+Sepal.Width,family=binomial,data=iris,control=list(100))
> summary(lm12.log)

Call:
glm(formula = V6 ~ Sepal.Length + Sepal.Width, family = binomial,
    data = iris, control = list(100))

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.3814  -0.4464   0.3818   0.5728   1.1168

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)   -1.3390     2.0128  -0.665    0.506
Sepal.Length    1.8212     0.2301   7.916 2.45e-15 ***
```

```

Sepal.Width    -2.7777      0.4371   -6.355 2.08e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 190.954  on 149  degrees of freedom
Residual deviance:  52.626  on 147  degrees of freedom
AIC: 58.626

Number of Fisher Scoring iterations: 1

```

上述代码在 `glm()` 函数中添加了用于控制迭代次数的参数 `control`, `control` 接受一个列表作为输入对象, 因此在上述代码中指定参数 `control` 的值为 `list(100)`, 即模型迭代 100 次。此时模型不再弹出警告信息, 使用 `summary()` 函数查看模型中的信息, 也得到了正常的结果。

观察 R 的返回结果, 可得 $p(V6=1|SL,SW)=\frac{\exp(-1.339\ 0+1.821\ 2\times SL-2.777\ 7\times SW)}{1-\exp(-1.339\ 0+1.821\ 2\times SL-2.777\ 7\times SW)}$, 其中, SL、SW 分别为 Sepal.Length、Sepal.Width 的简写。但常数项的 p 值大于 0.05, 因此需从模型中剔除常数项。

```

> exp(coef(lm12.log))
(Intercept) Sepal.Length Sepal.Width
0.26211694   6.17946697   0.06217937

```

查看 `lm12.log` 的优势比, 当 Sepal.Length 固定不变时, Sepal.Width 每增加一个单位, 因变量属于类别 1 的概率就增加 0.062 179 37 倍; 当 Sepal.Width 固定不变时, Sepal.Length 每增加一个单位, 因变量属于类别 1 的概率就增加 6.179 466 97 倍。显然, 因变量属于类别 1 的概率随着 Sepal.Length 的增大而增大, 随着 Sepal.Width 的增大而减小。

本节的模型都是二值 Logistic 模型, Logistic 模型对自变量的要求比较宽松, 哑变量、有序变量等都可以引入 Logistic 模型, 但 `glm()` 函数仅能处理因变量为二分类的情况, 当因变量的类别多于两种时, 最好的方法就是构建多个二值 Logistic 模型。将 `iris` 数据集中的 `Species` 变量作为因变量引入 `glm()` 函数时, `glm()` 函数会按照输入顺序处理数据, 即将 `setosa` 作为类别 0, 将其他类别作为类别 1 处理。

10.3 泊松回归分析模型

本节将讨论因变量既不是连续变量、也不是分类变量的情况。泊松分布是自然界中较为常见的离散分布之一, 本节将介绍服从泊松分布的常见情景, 以及如何使用 `glm()` 函数拟合泊松回归分析模型并检验模型是否显著。

10.3.1 拟合第一个泊松回归模型

泊松分布是自然界中较常见的离散分布之一，它是从二项分布延伸出的一种分布。但与只能取 0 或 1 的二项分布相比，泊松分布能取到的值有许多。

泊松分布通常用于排队模型或计数模型，每分钟内经过红绿灯的车辆数目、每分钟内公交车站中等待公交车的人数、每分钟接到电话的呼叫次数等，这些模型都属于泊松分布。在泊松分布模型中取值范围为从 0 到无穷大的整数，泊松分布的密度分布图如图 10.6 所示。

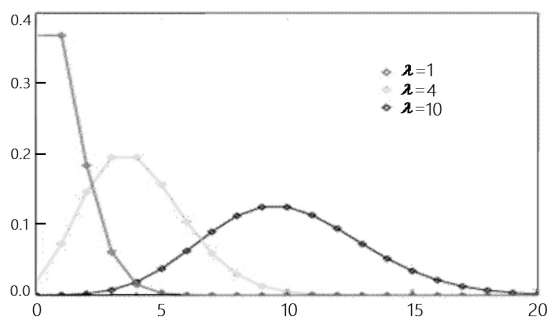


图 10.6 泊松分布的密度分布图

图 10.6 给出了泊松分布的参数 λ 分别取 1、4、10 时的密度分布曲线。 λ 取值越大，曲线的峰值就越低，起伏就越平缓，且密度分布的峰点随着 λ 的增大而逐渐右移。当 λ 取值为 10 时，泊松分布的密度曲线已经非常近似于正态分布，但正态分布是连续分布，曲线上的每一处都能取到，而泊松分布却是离散分布，仅能取到整数点，即图 10.6 中用小圆圈着重标出的点。

概括来讲，泊松分布用于研究单位时间内的计数问题，对服从泊松分布的因变量来说，一般线性回归模型不起作用，需要为其构建广义线性模型。`glm()` 函数同样提供了构建泊松回归模型的参数。

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4   67     5   1
2   36    118  8.0   72     5   2
3   12    149 12.6   74     5   3
4   18    313 11.5   62     5   4
5   NA     NA 14.3   56     5   5
6   28     NA 14.9   66     5   6
```

在正式构建模型前，首先来了解准备用来构建模型的原始数据。数据集 `airquality` 是一份关于纽约空气质量的数据，其中包含 `Ozone`（臭氧）、`Solar.R`（阳光）、`Wind`（风力）、`Temp`（温度）四个空气指标，以及 `Month`（月份）、`Day`（日期）两个时间指标。

上述代码查看了 `airquality` 中的前 6 行数据，其中 `Wind` 的取值不是整数，不能作为泊松回归模型中的因变量，不妨令 `Ozone` 作为泊松回归模型的自变量，研究 `Solar.R`、`Wind`、`Temp` 的波动是否会影响 `Ozone` 的值。

```

> lm.pos <- glm(Ozone~Solar.R+Wind+Temp,family=poisson,data=airquality)
> summary(lm.pos)

Call:
glm(formula = Ozone ~ Solar.R + Wind + Temp, family = poisson,
    data = airquality)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-5.5259  -2.0146  -0.5222   1.5665   9.2242

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.597270    0.195003   3.063  0.00219 **
Solar.R      0.002258    0.000208  10.857 < 2e-16 ***
Wind        -0.082384    0.005223 -15.773 < 2e-16 ***
Temp         0.042744    0.002055  20.803 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 2627.1  on 110  degrees of freedom
Residual deviance:  752.7  on 107  degrees of freedom
(42 observations deleted due to missingness)
AIC: 1344.8

Number of Fisher Scoring iterations: 4

```

上述代码使用 `glm()` 函数构建了泊松回归模型，其用法与构建 Logistic 模型时相似，函数中第一个参数指定 `Ozone` 为因变量，`Solar.R`、`Wind`、`Temp` 为自变量，参数 `family` 指定依据泊松分布构建模型，由于并未绑定 `airquality` 数据集，因此还需在 `glm()` 函数中指定参数 `data` 为 `airquality`，表明自变量和因变量从数据集 `airquality` 中取得。

利用 `summary()` 函数查看了模型中的具体信息，常数项和三个自变量系数的 p 值都小于 0.01，根据这些系数，可得方程 $\ln(\text{Ozone}) = 0.597\ 270 + 0.002\ 258 \times \text{Solar.R} - 0.082\ 384 \times \text{Wind} + 0.042\ 744 \times \text{Temp}$ 。这个方程显然是显著的。`Deviance Residuals` 显示方程的残差范围为 -5.5~9.2，但由于此时假设因变量服从泊松分布，因此残差检验在这里并不起作用。

```

> anova(lm.pos,test="Chisq")
Analysis of Deviance Table

Model: poisson, link: log

Response: Ozone

```

Terms added sequentially (first to last)

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			110	2627.1	
Solar.R	1	370.75	109	2256.4	< 2.2e-16 ***
Wind	1	1051.28	108	1205.1	< 2.2e-16 ***
Temp	1	452.40	107	752.7	< 2.2e-16 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

一些适用于 Logistic 模型的检验同样也适用于泊松回归模型。上述代码检验了模型 `lm.pos` 的似然比卡方。`Solar.R`、`Wind`、`Temp` 的模型偏差分别为 370.75、1 051.28 和 452.40，它们的 p 值都小于 0.001，即能对模型造成较大的偏差，适合引入泊松回归模型。

```
> exp(lm.pos$coefficients)
(Intercept)      Solar.R      Wind      Temp
1.8171505      1.0022608      0.9209186      1.0436709
```

泊松回归模型的自变量系数同样不能直接比较。上述代码查看了模型 `lm.pos` 中自变量系数的指数，该命令与命令“`exp(coef(lm.pos))`”具有相同的效果。观察 R 返回的结果，当 `Wind` 和 `Temp` 保持不变时，`Solar.R` 每增加一个单位，`Ozone` 将增加 1.002 262 8 倍；`Solar.R` 和 `Temp` 保持不变时，`Wind` 每增加一个单位，`Ozone` 将增加 0.920 918 6 倍；`Solar.R` 和 `Wind` 保持不变时，`Temp` 每增加一个单位，`Ozone` 将增加 1.043 670 9 倍。`Solar.R` 和 `Temp` 的增长会引起 `Ozone` 的增长，`Wind` 的增长则会引起 `Ozone` 的降低。

10.3.2 泊松回归模型的过散布检验

在泊松回归模型中分析人员通常希望因变量完美地服从泊松分布，与假设因变量服从正态分布不同，一个变量是否服从泊松分布由许多依赖变量决定。当影响因变量的因素全部包含在自变量中时，泊松回归模型能够完美地拟合出回归方程，而当影响因变量的因素并未完全纳入自变量时，泊松回归模型就需要调整为极大似然估计法，此时也称回归模型出现了过散布的情况。

```
> library(qcc)
> airquality2 <- na.omit(airquality)
> qcc.overdispersion.test(airquality2$Ozone, type="poisson")
```

Overdispersion test	Obs.Var/Theor.Var	Statistic	p-value
poisson data	26.30199	2893.219	0

程序包 `qcc` 提供了检验泊松回归模型是否出现过散布的函数 `qcc.overdispersion.test()`，上述代码中第 1 行代码加载了 `qcc` 程序包。

在使用 `head()` 函数查看 `airquality` 中的数据时可以发现其中包含空缺值，`glm()` 函数在构建模型时自动剔除了空缺值，但 `qcc.overdispersion.test()` 函数并不能自动剔除空缺值，

因此上述代码在正式检验过散布之前使用 `na.omit()` 函数将 `airquality` 中不含空缺值的样本挑了出来，并赋给了变量 `airquality2`，实际上，使用 `glm()` 函数为 `airquality2` 构建泊松回归模型将得到与模型 `lm.pos` 相同的结果。

上述代码中第 3 行代码使用 `qcc.overdispersion.test()` 函数检验了 `Ozone` 的过散布，函数中第一个参数指定 `Ozone` 为被检验对象，第二个参数 `type` 则指定检验的类型为泊松分布。观察 R 的返回结果，观测方差与理论方差之比为 26.301 99，其 p 值为 0，说明 `Ozone` 出现了过散布的情况，其实际分布的方差比泊松分布的方差更大，因此需要调整模型，使用准极大似然估计法构建泊松回归模型。

```
> lm2.pos <- glm(Ozone~Solar.R+Wind+Temp,family=quasipoisson,data=airquality)
> summary(lm2.pos)

Call:
glm(formula = Ozone ~ Solar.R + Wind + Temp, family = quasipoisson,
    data = airquality)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-5.5259  -2.0146  -0.5222   1.5665   9.2242

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.5972696  0.5368161   1.113  0.268367
Solar.R      0.0022582  0.0005726   3.944  0.000144 ***
Wind        -0.0823837  0.0143786  -5.730  9.36e-08 ***
Temp         0.0427442  0.0056564   7.557  1.48e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for quasipoisson family taken to be 7.578241)

Null deviance: 2627.1  on 110  degrees of freedom
Residual deviance: 752.7  on 107  degrees of freedom
(42 observations deleted due to missingness)
AIC: NA

Number of Fisher Scoring iterations: 4
```

使用准极大似然估计法构建泊松回归模型的函数书写格式与构建模型 `lm.pos` 并无太大不同，与构建 `lm.pos` 的代码相比，上述代码仅修改了参数 `family`，指定其为 `quasipoisson`，即使用准极大似然法构建泊松回归模型。

观察 R 的返回结果，与模型 `lm.pos` 相比，`lm2.pos` 的残差值、常数项、各自变量系数均未发生变化，但常数项和自变量系数的标准误差变大，它们的 p 值也发生了变化，此时三个自变量仍是显著的，但常数项的 p 值大于 0.05，不再显著，应将其从模型中剔除。

```
> lm3.pos <- glm(Ozone~Solar.R+Wind+Temp-1,family=quasipoisson,data=airquality)
```

```

> summary(lm3.pos)

Call:
glm(formula = Ozone ~ Solar.R + Wind + Temp - 1, family = quasipoisson,
    data = airquality)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-5.4468  -1.9747  -0.6663   1.5110   9.8865

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
Solar.R    0.0023005   0.0005866   3.922 0.000155 ***
Wind      -0.0718007   0.0108269  -6.632 1.36e-09 ***
Temp       0.0487310   0.0017805  27.370 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for quasipoisson family taken to be 7.80518)

Null deviance: 28457.42  on 111  degrees of freedom
Residual deviance:  762.02  on 108  degrees of freedom
(42 observations deleted due to missingness)
AIC: NA

Number of Fisher Scoring iterations: 4

```

上述代码在 `lm2.pos` 的基础上删去了常数项，并创建了模型 `lm3.pos`，观察 R 的返回结果，此时回归方程变为 $\ln(\text{Ozone}) = 0.002\ 300\ 5 \times \text{Solar.R} - 0.071\ 800\ 7 \times \text{Wind} + 0.048\ 731\ 0 \times \text{Temp}$ ，其中三个自变量系数都是显著的。

10.4 广义线性模型的交叉验证

在构建线性回归模型时，我们不仅关心方程整体及方程中每一个系数的可信程度，也关心模型的精度，即模型是否能够很好地拟合出数据的真实情况。在本章的 10.1 节中已经体现出了 R 方高的模型不一定是拟合得最好的模型，因此，检验模型的精度就是非常重要的一件事，而对于检验指标较少的 Logistic 模型和泊松回归模型来说，模型的精度检验更是意义非凡。

检验模型的精度至少需要一组训练集和一组测试集，训练集和测试集的真实值都是已知的，将训练集拿来构建模型，并使用模型预测测试集中样本的值，通过对比测试集中预测值和真实值的差别，即可知道模型的精度如何，显然，预测值和真实值越接近，模型拟合得越好，精度也越高。

当样本数据足够多时，可以随意拿出一小部分数据作为测试集，而不会影响训练集

构建模型。但当样本数据比较少时，随意拆分样本数据可能会导致训练集中数据过少，从而影响模型的精度。一个比较简单的解决办法是留一旁置法，即仅从样本数据中拿出一个样本作为测试集，其他数据全部作为训练集。这虽然保证了训练集中样本的数目，但也可能会影响检验的准确度和可信程度。

另一种较好的办法是交叉验证法，它将样本数据划分为 k 份样本数目相同的数据集，首先将第一份数据作为测试集，将其他 $k-1$ 份数据作为训练集，训练数据并验证模型的精度；然后将第二份数据作为测试集，将其他 $k-1$ 份数据作为训练集，训练数据并验证模型的精度……如此循环 k 次，得到 k 份有关模型精度的检验结果，最后综合这 k 份结果，得到最终结果。

由于交叉验证要求每一份数据集中样本的数目都相同，因此，全体样本数目必须恰好是 k 的整倍。程序包 `boot` 提供了有关交叉验证的函数，不妨以 Logistic 模型为检验对象，验证其准确度是否足够好。

```
> iris[,6] <- c(rep(0,50),rep(1,100))
> iris$V6 <- as.factor(iris$V6)
> lm.log <- glm(V6~Sepal.Length,family=binomial,data=iris)
```

上述代码首先在 `iris` 数据集中添加了一列变量 `V6`，其前 50 个值为 0，后 100 个值为 1，然后用 `as.factor()` 函数将 `V6` 转换为因子类型的变量。最后一行代码使用 `glm()` 函数构建了模型 `lm.log`，函数中的参数 `family` 为 `binomial`，表示这是一个 Logistic 模型，该模型在 10.2 节已出现过，它用于判断新样本是否属于 `setosa` 类，当新样本的变量 `V6` 的预测值为 0 时，新样本便属于 `setosa` 类；当新样本的变量 `V6` 的预测值为 1 时，新样本便不属于 `setosa` 类。

```
> library(boot)
> cv.glm(data=iris,glmfit=lm.log,K=5)$delta
[1] 0.0769443 0.0766430
> sum(lm.log$residuals^2)/lm.log$df.residual
[1] 5.944237
```

上述代码首先加载了 `boot` 包，第 2 行命令使用 `cv.glm()` 函数交叉验证了 `lm.log` 的精确度，其中，第一个参数 `data` 指定数据来源为 `iris`，第二个参数 `glmfit` 指定被检验的模型为 `lm.log`，第三个参数 `K` 则指定交叉验证为 5 折交叉验证，由于 `iris` 中共有 150 个样本，故 `K` 取 5 是可行的。当样本数目为质数，或不太好划分时，可以使用 `set.seed()` 函数允许交叉验证重复抽取样本。

在第 2 行代码的末尾使用符号 `$` 提出了 `cv.glm()` 函数的结果中的 `delta` 元素，R 同时返回了交叉验证的预测误差 0.076 944 3 和留一旁置法的预测误差 0.076 643 0，第三行代码则计算了基于全体样本所建模型的预测误差，该值为 5.944 327，是一个比交叉验证的预测误差和留一旁置法的预测误差大得多的数值，这说明模型 `lm.log` 的预测误差比真实误差乐观得多，尽管模型 `lm.log` 在样本数据上拟合得很好，但它在新样本上的表现不会太好。

第 11 章 R 中的时间序列模型

本章将介绍如何用 R 构建时间序列模型。时间序列模型是一类广泛应用于金融领域和自然社会科学领域的模型，本章将讨论其基本性质，并使用 R 构建平稳的时间序列模型和非平稳的时间序列模型，对希望从事金融行业的读者来说，本章尤其重要。

11.1 将数据转换为时间序列格式

时间序列模型具有很多特别的性质，本节将讨论如何利用 `ts()` 函数或 `zoo()` 函数将矩阵转换为时间序列，也将涉及月数据和日数据的不同处理方法，并探究一些时间序列的特点。

11.1.1 使用 `ts()` 函数转换数据格式并绘制时间序列曲线

时间序列描述了在时间维度上连续发生的事情，预知未来是一种极具商业价值的行为，如果我们能在彩票中奖号码还未揭晓前就能预知中奖号码，那么我们都会成为百万富翁，而彩票公司就会破产。时间序列模型所要做的就是使用过去和现在的数据预测未来的数据，要做到这一点，首先要获取正确格式的数据。

```
> WorldPhones
      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
1951  45939  21574 2876   1815   1646    89    555
1956  60423  29990 4708   2568   2366   1411   733
1957  64721  32510 5230   2695   2526   1546   773
1958  68484  35218 6662   2845   2691   1663   836
1959  71799  37598 6856   3000   2868   1769   911
1960  76036  40341 8220   3145   3054   1905  1008
1961  79831  43173 9053   3338   3224   2005  1076
```

数据集 `WorldPhones` 中一共存放了 7 条有关手机拥有量的数据，上述代码查看了其中的全体信息。这是一个规规矩矩的矩阵，它存放了七大洲在 7 个年份内的手机拥有量，其中矩阵的列是有关洲的变量，矩阵的行是有关年份的变量。观察这 7 个年份，它们分别是 1951 年及 1956~1961 年之间的每一年，尽管此时 `WorldPhones` 看起来是一份非常标准的时间序列，但此时它仍仅为一个矩阵，而 7 个年份也不过是每一列的列名，此时并不能对 `WorldPhones` 应用有关时间序列的函数。

```
> WorldPhones <- WorldPhones[-1,]
> WorldPhones
```

	N.Amer	Europe	Asia	S.Amer	Oceania	Africa	Mid.Amer
1956	60423	29990	4708	2568	2366	1411	733
1957	64721	32510	5230	2695	2526	1546	773
1958	68484	35218	6662	2845	2691	1663	836
1959	71799	37598	6856	3000	2868	1769	911
1960	76036	40341	8220	3145	3054	1905	1008
1961	79831	43173	9053	3338	3224	2005	1076

上述代码删去了 `WorldPhones` 中的第 1 行数据，即 1951 年的数据。再次查看其中的全体数据，此时仅剩下了 6 条数据，它们的年份跨度是连续的，这一步是接下来转换时间序列格式的准备工作。

```
> WorldPhones <- ts(WorldPhones,start=1956,end=1961,frequency=1)
> WorldPhones
Time Series:
Start = 1956
End = 1961
Frequency = 1
      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
1956  60423  29990 4708   2568   2366   1411    733
1957  64721  32510 5230   2695   2526   1546    773
1958  68484  35218 6662   2845   2691   1663    836
1959  71799  37598 6856   3000   2868   1769    911
1960  76036  40341 8220   3145   3054   1905   1008
1961  79831  43173 9053   3338   3224   2005   1076
> plot(WorldPhones)
```

上述代码中的第 1 行命令使用 `ts()` 函数将 `WorldPhones` 转换为 `ts` 格式，`ts` 格式是最基础的一种时间序列格式，其他时间序列格式还有 `zoo`、`zooreg` 等。在上述代码中，`ts()` 函数一共有 4 个参数，第一个参数指定被转换的对象，第二个参数 `start` 指定时间索引从 1956 年开始，第三个参数 `end` 指定时间索引到 1961 年结束。

最后一个参数 `frequency` 则指定时间索引的频率为 1，即每一年中只有一个索引，该参数也可设置为其他的数值，当它设置为 4 时，即在每一年中按照季度给出索引，当它设置为 12 时，即按照月份给出索引。参数 `frequency` 生成的索引数目过少时，`ts()` 函数会删掉被转换数据中盛不下的数据；参数 `frequency` 生成的索引数目过多时，`ts()` 函数则会循环使用数据样本。

再次查看 `WorldPhones` 中的数据，`ts` 格式的数据与矩阵格式的数据并无太大差别，只是在表格顶部添加了一些数据格式信息。此外，矩阵的每一行的行名称也从数值型名称变为了时间索引，在绘图或计算时，时间索引都会起到作用。

利用 `plot()` 函数绘制 `WorldPhones` 的时间序列图，其结果如图 11.1 所示，R 为 `WorldPhones` 中的 7 列数据逐一绘出了 7 张曲线图，横坐标则将时间索引标注了出来，观察这 7 张曲线图，每一个洲的手机拥有量都随时间的增加而增长，曲线图的纵轴坐标提供了不同洲间增幅的对比依据。

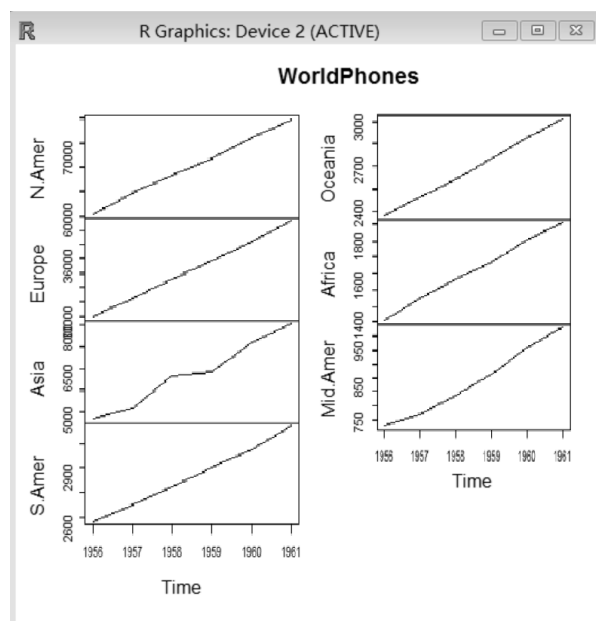


图 11.1 绘制 WorldPhones 的时间序列图

11.1.2 使用 zoo() 函数转换数据格式并绘制时间序列曲线

在 WorldPhones 的例子中我们转换了一个标准的时间序列，但真实情况有时比 WorldPhones 要复杂得多，比如逐日记录数据的时间序列。

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> air.ts <- ts(airquality)
> plot(air.ts)
```

数据集 `airquality` 是一份有关纽约空气质量的数据，上述代码首先使用 `head()` 函数查看了它的前 6 行数据，与 `WorldPhones` 不同，`airquality` 是一份逐日记录的数据，实际上它记录了 5 月 1 日~9 月 30 日之间每一天的空气质量数据。它同样能够被 `ts()` 函数转换，上述代码的第 2、3 行命令将 `airquality` 转换为 `ts` 格式的数据 `air.ts`，并使用 `plot()` 为其绘制了时间序列曲线，其结果如图 11.2 所示。

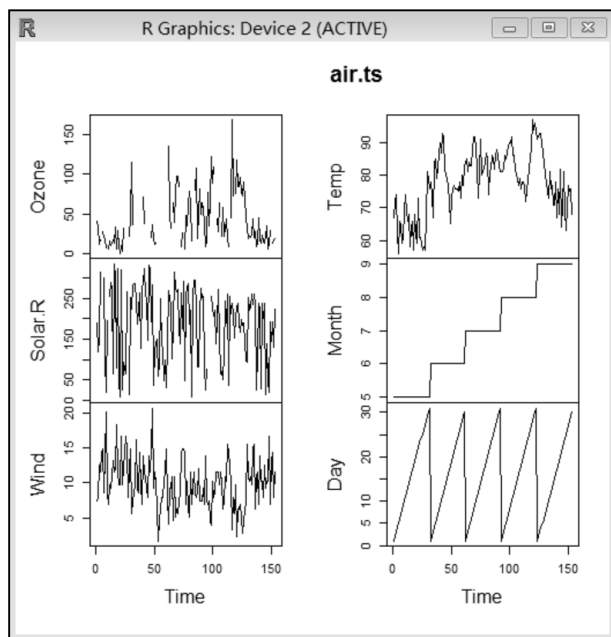


图 11.2 为 air.ts 绘制时间序列图

观察图 11.2, R 一共绘出了 6 张曲线图, 其中 Month 和 Day 是我们所不需要的, 应该从 air.ts 中删去。此外图 11.2 的横轴并不是时间索引, 没有时间索引做横轴后, 显然, 时间序列图的含义将变得含糊不清。但我们并不能像例子 WorldPhones 中那样在 ts() 函数中使用 start、end、frequency 等参数生成时间索引, 显然, 由于每个月的天数并不一致, 因此 frequency 不是一个固定的值, ts() 函数没办法处理 airquality 这样的日数据。

```
> time <- as.Date('1973-05-01')+c(0:152)
> tail(time)
[1] "1973-09-25" "1973-09-26" "1973-09-27" "1973-09-28" "1973-09-29"
[6] "1973-09-30"
> library(zoo)
> air.ts <- zoo(air.ts,time)
```

为了解决 ts() 函数无法解决的问题, 需要首先生成一个时间索引向量, 再将时间索引添加到时间序列数据中。上述代码的第 1 行代码使用 as.Date() 函数生成了一个 Date 格式的时间 1973-05-01, airquality 中的数据采集于 1973 年, 因此这个时间显然是时间索引的起点。

从 5 月到 9 月这 5 个月份共有 153 天, 因此令 1973-05-01 与从 0 到 152 的序列逐一相加, 就得到了从 5 月到 9 月的全部时间索引。上述代码中 tail() 函数查看了 time 中的最末 6 个元素, 显然 as.Date() 函数在每一个月中的天数数完时就自动将时间索引推移至下一个月。通过 as.Date() 函数与 c() 函数的组合也能够生成跳跃的时间指标, 只需将 c() 函数中的数字序列指定为跳跃序列即可。

将时间索引和时间序列数据组合起来的函数为 `zoo()` 函数，它存在于 `zoo` 包中，上述代码的第 3、4 行命令首先加载了 `zoo` 包，然后使用 `zoo()` 函数将时间序列数据 `air.ts` 和时间索引 `time` 组合起来并再次赋给了 `air.ts`。`zoo()` 函数同样也能应用于矩阵数据，命令“`zoo(airquality,time)`”能达到与上述代码相同的效果。

```
> air.ts <- air.ts[, -c(5:6)]
> head(air.ts)
      Ozone Solar.R Wind Temp
1973-05-01   41    190  7.4   67
1973-05-02   36    118  8.0   72
1973-05-03   12    149 12.6   74
1973-05-04   18    313 11.5   62
1973-05-05   NA     NA 14.3   56
1973-05-06   28     NA 14.9   66
> plot(air.ts)
```

上述代码首先删去了 `air.ts` 中的第 5、6 列变量，即有关月份和日期的变量。第 2 行代码中的 `head()` 函数查看了 `air.ts` 中的前 6 行数据，此时数据中只留下了我们感兴趣的 4 项空气指标，每一条数据的左边也添加了按照日期排列的时间索引，此时 `air.ts` 已经转换成理想的时间序列数据，使用 `plot()` 函数再次为其绘制时间序列图，其返回结果如图 11.3 所示。

观察图 11.3，`Ozone`、`Solar.R`、`Wind`、`Temp` 均呈现出不规律的波动，难以从图 11.2 中直接看出这 4 项指标是否会随时间的变化而有所变化。此时仍需进一步处理时间序列数据，才能得出有用的结论。

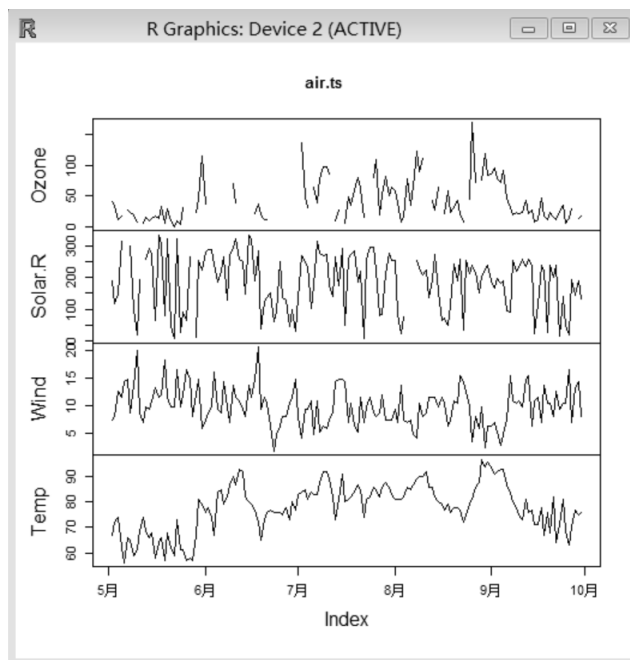


图 11.3 添加时间索引后的时间序列图

11.2 分解时间序列并检验时间序列的自相关性

时间序列中往往暗含一些周期规律，分解时间序列能从时间序列中剥离出这些规律。本节将讨论如何使用不同的方法分解时间序列，以及如何检验时间序列的自相关性。

11.2.1 使用经典方法分解时间序列

如果想要预测时间序列在未来的取值，那么时间序列的规律性越明显就越好预测。比如对于温度来说，每年的夏天都比冬天气温更高，因此有理由认为明年夏天的气温还会高于明年冬天。通常认为一个时间序列由长期趋势因素（T）、季节波动因素（S）、周期波动因素（C）和随机波动因素（I）4 部分组成。

长期趋势指时间序列的长期变化趋势，以气温为例，最近全球变暖，因此气温的长期趋势为上升趋势；季节波动则指时间序列随时间呈现的波动，气温以一年为一个波动周期，月亮的圆缺则以一个月为一个波动周期；周期波动因素是和长期趋势紧密相连的一个因素，长期趋势在每一季节都会有所变化，周期波动衡量的则是长期趋势在每一季节中的变化；随机因素则是时间序列的随机波动。

将长期趋势、季节波动、周期波动和随机波动叠加起来，即可得到最终的时间序列模型。这 4 种因素的叠加模型有加法模型、乘法模型和混合模型，其中加法模型认为 4 种因素的和即为真实结果；乘法模型认为 4 种因素的积即为真实结果；混合模型则认为长期趋势、季节波动、周期波动的积加上随机波动得到的结果为真实结果。

经典分解法使用移动平均值 MA 求解这 4 种因素的值。以三天为例，今天的平均值即昨天、前天和今天三天观测值的均值，明天的均值则为今天、明天和后天三天观测值的均值，以此类推，直至最后一天，但在三天移动平均中，时间序列的第一天和最末一天推算不出对应的平均值。

移动平均值的平均天数可以任意选择，通常选择小于 10、大于 1 的奇数，天数选择得越小，平均值就越容易受到异常值的影响；天数选择得越大，时间序列的首尾两端无法计算移动平均值的样本则越多，移动平均序列越短。

移动平均序列中同时包含了 T 和 C，使用真实值减去移动平均值，即可得到 S 和 I 的和，由于随机因素相加的和总为零，因此对 MA-T-C 的结果再取移动平均值，即可分离出 S，而 MA-T-C-S 的值，即为 I。

R 中按照经典分解方法分解时间序列的函数为 `decompose()` 函数，它存在于基础包中，只需直接调用该函数即可。

```
> AP.dec <- decompose(AirPassengers)
> plot(AP.dec)
```

上述代码使用 `decompose()` 分解了时间序列 `AirPassengers`，这是一个关于飞机托运行李重量的时间序列，其分解结果存放在 `AP.dec` 中，利用 `plot()` 函数绘制分解后的时间序列图，图 11.4 是 R 的返回结果。

图 11.4 由 4 个曲线图 `observed`、`trend`、`seasonal` 和 `random` 组成，其中，`observed` 是

根据时间序列 `AirPassengers` 绘出的曲线图，`AirPassengers` 中的数据有一个明显的周期波动，且具有明显向上的趋势；`trend` 绘出了 `AirPassengers` 的长期趋势，这是一条向上倾斜的曲线，它反映了 T 与 C 的和，根据 `trend` 中的值可拟合出一条回归直线，该直线即为 T ，而 `trend` 与 T 的差即为 C 。通常情况下无须将 T 和 C 都计算出来。

`seasonal` 绘出了 `AirPassengers` 的季节趋势，它显然以年为周期，每个周期内都有一个高峰和两个次高峰，`AirPassengers` 呈现出非常明显的季节规律；`random` 绘出了 `AirPassengers` 的随机波动，`AirPassengers` 在 1952~1956 年内随机波动较小，其他年份的随机波动则较强。

`AirPassengers` 的随机波动主要落于 $-40 \sim 50$ 之间，该范围明显小于 `seasonal` 和 `trend` 的分布范围，因此 `AirPassengers` 具有相当明显的规律，依据其长期趋势和季节趋势，即可较为合理地预测 `AirPassengers` 的未来值。图 11.4 所绘的是依据加法模型分离出的结果，在 `decompose()` 函数中设定参数 `type` 为 `multiplicative`，也可根据乘法模型分离时间序列。

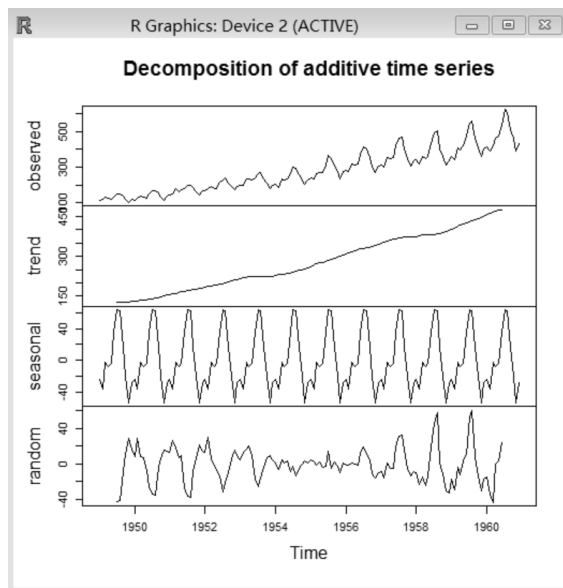


图 11.4 使用经典法分解 `AirPassengers` 序列

11.2.2 使用 STL 方法分解时间序列

经典分解法具有通俗易懂、易于计算的优点，但它同样也具有缺点，其中最突出的缺点在于经典分解法不能顾及时间序列首尾两端的样本，移动平均的天数取得越多，不能顾及的样本点就越多；经典分解法假设季节型因素是恒定的，当时间序列的年份跨度较大时，季节因素很可能发生变化，从而违反这一假设；此外经典分解对异常值也过于敏感。

基于上述缺点，如今经典分解法已不是最常用的时间序列分解方法，STL 分解法具有比它更优秀的性质。STL 分解法又称为局部加权回归散点平滑法，它发明于 1990 年，是如今最流行的时间序列分解方法。STL 分解法的核心是 Loess 算法，它同时使用了一

个内循环和一个外循环，内循环使用周期平滑、滤波处理等 6 个步骤分解出季节因素和长期趋势因素，外循环则引入一个权重，用于消除样本数据中异常值的影响。

R 的基础包中同样提供了使用 STL 分解法分解时间序列的 `stl()` 函数，只需直接调用该函数即可。

```
> AP.stl <- stl(AirPassengers,t.window=13,s.window=7)
> plot(AP.stl)
```

上述代码使用 `stl()` 函数再次分解了 `AirPassengers` 序列，在 `stl()` 函数中，指定参数 `t.window` 为 13，它用于计算长期趋势的时间窗宽度，指定参数 `s.window` 为 7，它用于计算季节趋势的时间窗宽度，这两个参数都必须指定，且都应当是奇数。`plot()` 函数再次绘制了分解后的时间序列图，R 的返回结果如图 11.5 所示。

观察图 11.5，它同样由 `data`、`seasonal`、`trend`、`remainder` 4 个图形组成，其中，`data` 与图 11.4 中的 `observed` 一致，绘出了被分解数据的时间序列图；`seasonal` 则绘出了 `AirPassengers` 的季节趋势图，与图 11.4 中的季节趋势图相比，图 11.5 给出的季节趋势图随着时间的增长，波动程度变得越来越剧烈。

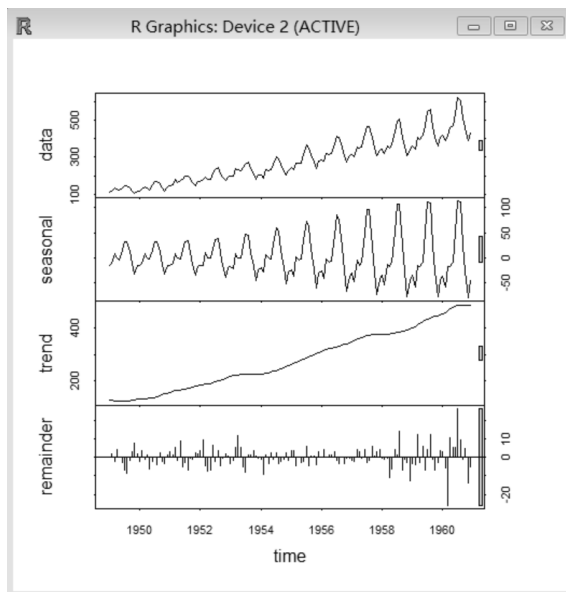


图 11.5 使用 STL 方法分解 `AirPassengers` 序列

`trend` 绘出了 `AirPassengers` 的长期趋势图，该图与图 11.4 并无太大差别；`remainder` 则绘出了随机因素，它由一条横线和位于横线上的一些竖线构成，每一条竖线对应一个样本点，竖线越长，对应样本点的随机波动就越大，竖线的方向则表明了样本点波动的方向。

图 11.5 还在每一个图形的右侧画出了一条长条，它们代表了同样大小的数值，其长度反映了图形的量纲，长条越长，量纲就越小。显然 `remainder` 的量纲是最小的，它的波动范围即最小。`seasonal` 的量纲比 `trend` 略小，说明 `seasonal` 在一个较小的范围内波动。

`stl()` 函数中 `t.window` 参数和 `s.window` 参数的大小直接影响最终的分离结果，`s.window`

参数的值也可指定为 `periodic`，表明用均值平滑季节性子列计算季节趋势，并根据数据自动调整时间窗宽度，是比较合理的方法。

与经典分解法相比，STL 方法具有稳健性好、可处理任何季节性（而不只是以年或以月为周期的季节性）、可控制平滑程度等优点，但它只能应用于加法模型，对于乘法模型来说，需对样本点取对数，使其变为加法模型，再在分解完毕后将结果逆变换回去，得到最终结果。

11.3 探究时间序列的自相关性

研究时间序列的唯一目的是找出其中的规律并用于预测未来趋势。显然，只有包含了较明显的规律的时间序列才是分析人员感兴趣的内容，否则将难以预测未来。因此，探究时间序列是否包含明显规律是在建模前非常重要的一项准备工作。

11.3.1 使用月图和季度图探究自相关性

分解时间序列一方面展示了时间序列中的长期趋势、季节性波动和随机波动，反映出时间序列的规律；一方面也为时间序列的预测作出了准备。除绘制分解后的时间序列图外，R 同样提供了一些其他的绘制函数，它们能够从各种角度绘出不同的图形，全方位地反映时间序列中的规律。

```
> monthplot(AirPassengers)
> seasonplot(AirPassengers)
```

R 中的基础包提供了绘制月图和季度图的 `monthplot()` 函数和 `seasonplot()` 函数，上述两条代码分别调用了这两个函数，为 `AirPassengers` 绘出了月图和季度图，R 的返回结果如图 11.6 所示，图中左边的图片为 `AirPassengers` 的月图，右边的图片为 `AirPassengers` 的季度图。

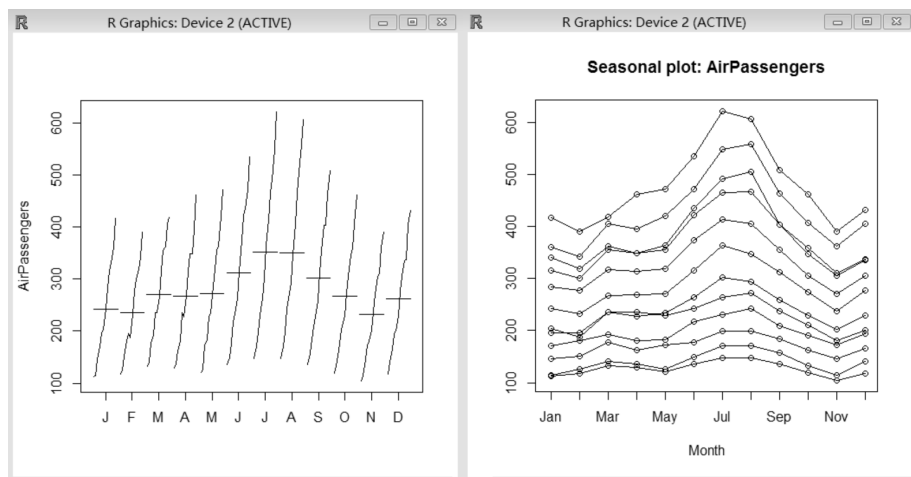


图 11.6 绘制 `AirPassengers` 的月图和季度图

观察 `AirPassengers` 的月图，它按照月份绘出了 `AirPassengers` 中的数据。`AirPassengers` 中存放了 1949~1960 年 12 年的数据，每一年都有 12 个月份的数据。月图将这些数据分为 12 组，每一组中存放了一个月的全部数据，比如第一组内存放了 1949~1960 年所有一月份的数据。

在图 11.6 所示的月图中，每一个月的数据都呈上升状态，显然时间靠后的年份内的某个月总比时间靠前的年份内的相同月具有更多的飞机行李。月图中还在每一组中绘出了一条小横线，用以标出均值的位置，这些小横线如果基本处于同一水平，则说明时间序列不随月份的变化而波动，图 11.6 中的月图均值明显在七、八月的时候出现了一个小高峰，说明 `AirPassengers` 的值明显在七、八月较大。

季度图与月图相似，它虽然名为季度图，但实际上也使用了 12 个月份的月数据用以绘图。观察图 11.6 中的季度图，它由 12 条折线构成，每条折线都反映了某一年内的 `AirPassengers` 数值，这 12 条曲线的位置逐年增高，反映出 `AirPassengers` 中上升的长期趋势，而曲线中的相同波动则反映了时间序列随月份变化而具有的波动规律。

在上述代码中并未指定更多的参数，因此绘出的图形仅具有基本效果。对于月图和季度图来说，最重要的是图标信息，月图的默认横坐标仅用首字母来代表 12 个月份，不妨指定参数 `xaxt` 为 `n`，去掉默认的横坐标，再使用 `axis()` 函数添加新的坐标轴。季度图则需为其添加每条折线的年份名称，当季度图中的折线较密集时，可以设置折线的颜色以便区分。

11.3.2 使用散点图探究自相关性

观察时间序列是否包含某种规律的另一种好方法是研究时间序列的自相关系数，自相关系数关心的是时间序列现在的数据和过去的的数据是否具有相关性，如果现在的数据与过去的的数据具有相关性，就说明此二者间具有联系，从而能够根据过去的的数据推测现在的数据，类似地，也能够根据现在的数据推测未来的数据。

```
> library(forecast)
> Acf(AirPassengers)
```

程序包 `forecast` 提供了绘制自相关系数的 `Acf()` 函数，上述代码首先加载了 `forecast` 程序包，然后调用 `Acf()` 函数为 `AirPassengers` 绘制了自相关系数图，R 的返回结果如图 11.7 所示。

观察图 11.7，它一共给出了 24 个自相关系数，分别给出了全体 `AirPassengers` 时间序列和与其相差 1~24 个数据的 `AirPassengers` 时间序列的相关系数，随着相差月份的增加，自相关系数从 0.9 逐渐滑落，并在相差 7 个月份后滑落到 0.65 的位置，在相差 7 个月份后，自相关系数又随着相差月份数的增加逐渐增长，在相差一年时恰好到达另一个高峰。这种自相关系数的变化以一年为一个周期，并随着周期的增加而逐渐降低，即数据的时间间隔越大，自相关系数就越小。

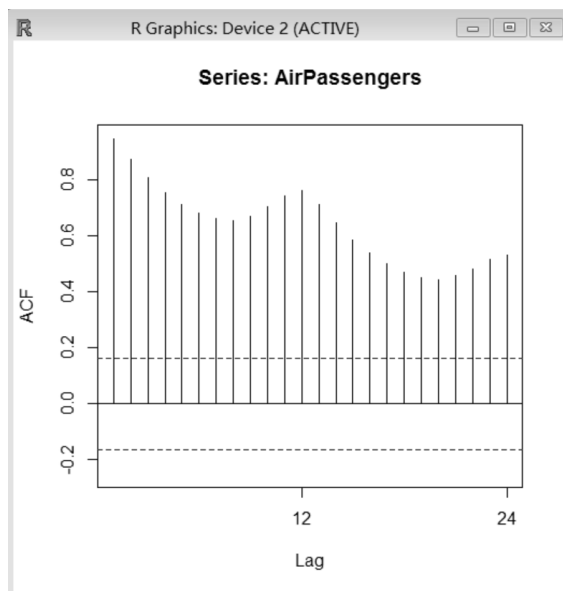


图 11.7 绘制 AirPassengers 的自相关系数图

```
> lag.plot(AirPassengers, lags=4)
```

自相关散点图是一种能够直观反映数据的自相关性的图形，R 中用于绘制自相关散点图的是 `lag.plot()` 函数，它存在于基础包中，上述代码调用了该函数，并为 `AirPassengers` 绘出了自相关散点图。其返回结果如图 11.8 所示。

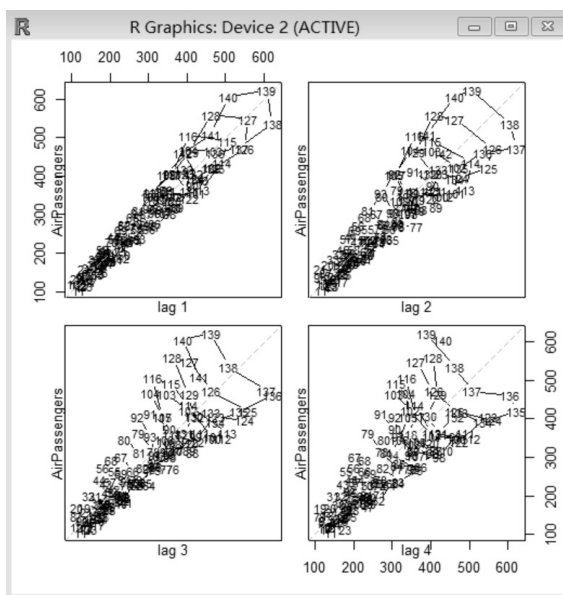


图 11.8 绘制 AirPassengers 的自相关散点图

自相关散点图绘制的是一种“时差版”的散点图，在上述代码中指定参数 `lags` 为 4，即绘制 4 张散点图，由于并未指定其他参数，因此此时默认的时差为 1、2、3、4。在

AirPassengers 中共有 144 个数据，图 11.8 中的 lag1 绘出的是这 144 个数据全体；lag2 绘制了第 2~144 个数据；lag3 绘制了第 3~144 个数据；lag4 则绘制了第 4~144 个数据。

尽管这 4 张散点图中的数据个数依次减少，但每张图中的散点都是从 1 开始标号的，因此，这 4 张图中标号相同的点并不对应同一个观测样本。观察图 11.8，这 4 张图片中的样本点都随着标号的增加而增长，而随着标号的增加，样本点分布的范围也将变大，即数据的波动将变得明显起来。

```
> lag.plot(AirPassengers[1:50], lags=4, set.lags=c(1, 13, 25, 37))
```

图 11.8 中的数据较为繁多，除能表现数据的长期趋势外，并不能表现更多的季节性趋势。上述代码再次调用 lag.plot() 函数，指定函数的绘制范围为 AirPassengers 的前 50 个数据，并增加了参数 set.lags 指定“时差”的大小为 12、24、36，即以一年为一个时差单位。

图 11.9 为 R 的返回结果。观察图 11.9，随着“时差”的增大，4 张散点图的数据样本有明显的锐减。这 4 张散点图都呈现向右上角倾斜的“∞”形状，散点图 lag37 中仅有一个“∞”形状，lag1 则由好几个“∞”形状叠加而成，显然，散点图形状的相似说明了 AirPassengers 中的数据有明显的季节性趋势，其波动的周期为年。

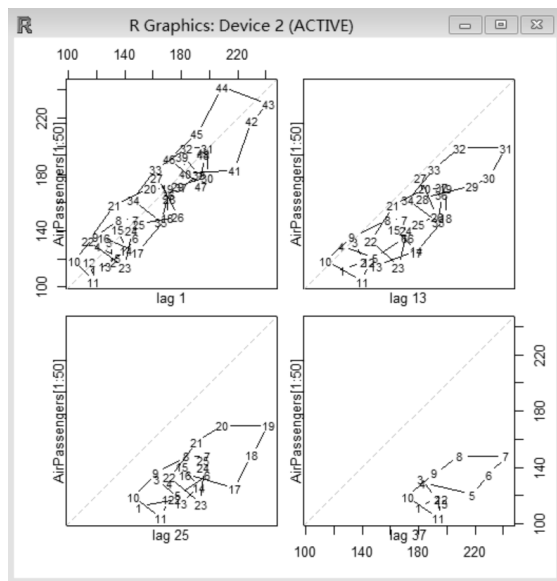


图 11.9 优化 AirPassengers 的自相关散点图

11.4 构建时间序列并预测

本节将讨论时间序列分析的最后一部分内容，即如何利用过去的的数据预测未来。如今成熟的时间序列模型主要有指数平滑模型、自回归模型和移动平均模型等，本节将讨论单纯预测等基础模型，并在此基础上讨论指数平滑模型和 ARIMA 模型等。

11.4.1 均值预测、单纯预测和漂移

均值预测、单纯预测和漂移是最简单的三种时间序列预测模型，本节在数据集 Nile 上应用了这三种模型。Nile 是一份有关河流长度的时间序列，它记录了从 1871 年到 1970 年这 100 年内每一年的河流长度。

```
> summary(Nile)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  456.0   798.5   893.5   919.4  1032.0  1370.0
> plot(Nile)
```

上述代码查看了 Nile 的摘要，它是一个数值型的向量，均值比中位数略大，上四分位数和下四分位数与中位数的差均为 100 左右，两个最值与中位数的差均为 450 左右。函数 plot() 绘出了 Nile 的曲线图，其结果如图 11.10 所示。

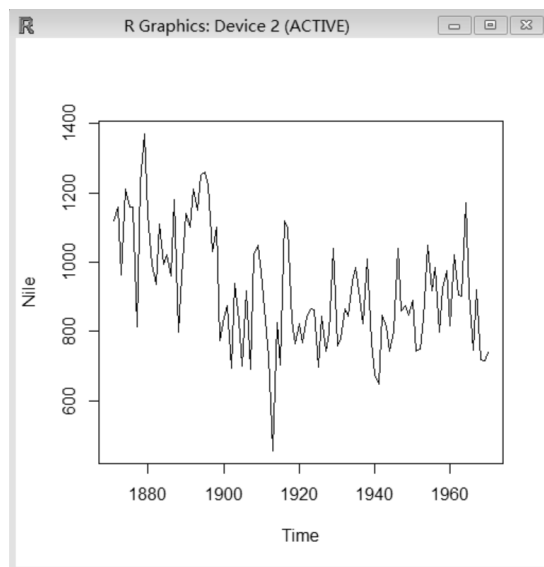


图 11.10 绘制 Nile 数据的时间序列图

观察图 11.10，时间序列曲线呈现剧烈的、无规律的波动，在 1900 年之前，Nile 基本在 1 000 上下来回波动，在 1900 年之后，Nile 的均值有所下降，在 800 左右来回波动。从数据的平稳性来看，Nile 的均值长期稳定在一个较小的范围内，因此 Nile 可以视为一份平稳时间序列，而 Nile 中仅有年度数据，而没有月度数据，因此无法对 Nile 进行时间序列分解，此时对 Nile 构造简单模型是探索数据结构的必要方法。

```
> library(forecast)
> Nile.f <- meanf(Nile,10)
> plot(Nile.f)
```

本节所应用的函数都存在于程序包 forecast 中，上述代码首先加载了该包，并在第 2 行命令中调用了用于构建均值预测模型的 meanf() 函数，并为 Nile 构建了均值预测模型，

`meanf()` 函数的第二个参数 10 表明向后预测 10 个数据, 即预测 1971~1980 年的河流长度。

均值预测模型使用时间序列的均值作为未来的预测值, 不难理解, 既然时间序列在过去的一段时间内都在某个固定的均值附近波动, 那么有理由认为, 在未来的一段时间内它还会在这个均值附近波动, 波动范围可由时间序列在过去时间段内的波动范围给出。上述代码同样使用 `plot()` 函数绘出了均值预测模型的预测结果, 其结果如图 11.11 所示。

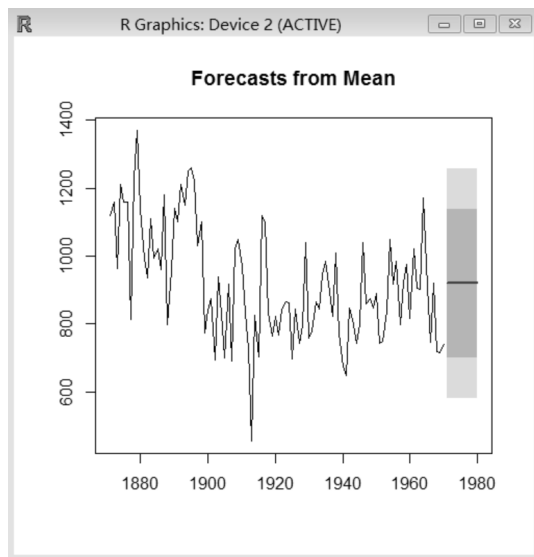


图 11.11 为时间序列 Nile 构建均值预测模型

观察图 11.11, 图中用一条蓝色的粗线标出了预测均值的位置, 并用深浅不一的蓝色区域标出了不同置信区间的预测值, 显然, 蓝色的粗线位于全部历史数据的均值位置, 而两个蓝色区域则能够框出大部分历史数据, 如果 Nile 的值在未来还会继续平稳持续下去, 那么它有很大概率会围绕蓝色粗线波动, 并落在蓝色区域中。

```
> Nile.n <- naive(Nile,10)
> plot(Nile.n)
```

单纯预测模型是另一种基本的时间序列预测模型, 上述代码同样调用了 `forecast` 程序包中的 `naive()` 函数构建了单纯预测模型, 并为数据集 Nile 预测了 1971~1980 年的数据。函数 `plot()` 再次绘出了预测值, R 的返回结果如图 11.12 所示。

单纯预测模型使用模型的最后一个值作为后续时间点的预测值, 即它认为 1971~1980 年 Nile 的值都是 1970 年 Nile 的值。图 11.12 中, 在 1971 年后绘出了一条蓝色直线, 这条直线上的值与 1970 年的值相同。图 11.12 同样也绘出了一个置信区间, 随着时间的增加, 该区间也逐渐向两端扩散。在单纯预测模型中, 置信区间的计算使用了自回归移动平均和随机游走等模型, 置信区间的参考价值远大于预测值的参考价值。

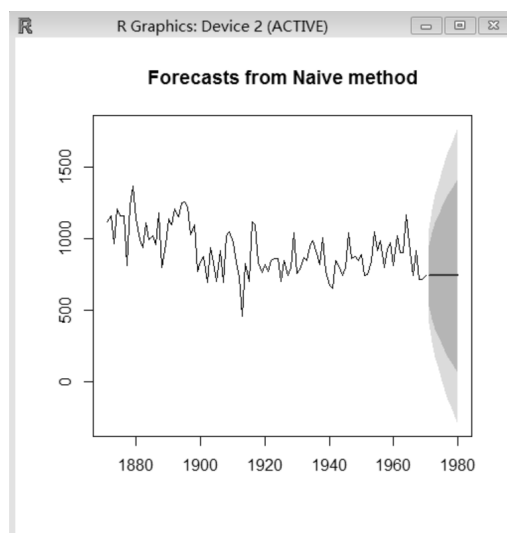


图 11.12 为时间序列 Nile 构建单纯预测模型

```
> Nile.r <- rwf(Nile,10,drift=T)
> plot(Nile.r)
```

漂移模型是最后一种常用的简单预测模型，上述代码使用 `rwf()` 函数构建了漂移模型，并同样向后预测了 10 年的 Nile 值，利用 `plot()` 函数绘制了预测结果图，图 11.13 是 R 返回的最终结果。

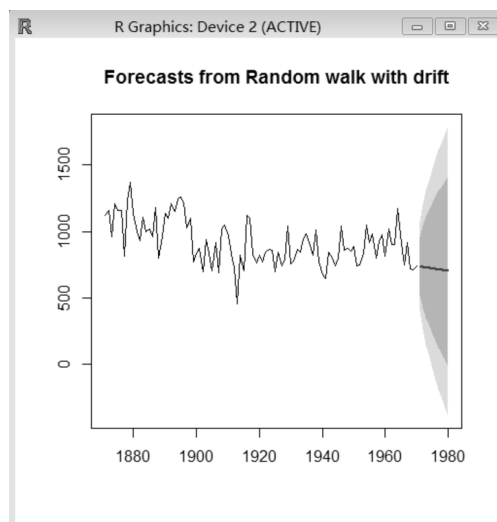


图 11.13 为时间序列 Nile 构建漂移模型

观察图 11.13，漂移模型画出的预测值同样附带一个置信区间。漂移模型认为预测值会随着时间的增加而逐渐变化，变化量与历史数据的平均变化量相同。从直观意义上理解，在图形中将历史数据的第一个点和最后一个点用一条线连接起来，并将线向未来延伸，即为漂移模型的预测值。

均值预测、单纯预测和漂移模型都可直接用于预测一些简单的、平稳的时间序列，但其预测结果通常并不被直接应用，而是用于与其他模型进行对比，显然，这些简单的模型对数据的预测准确度是最低标准，我们总希望最终的模型表现要优于这些简单模型。

11.4.2 不考虑长期趋势和季节波动的简单指数平滑

指数平滑法是介于均值预测法和单纯预测法之间的一种预测方法，均值预测法认为每一个历史数据都同等重要，根据全体历史数据计算出的均值是未来的预测值，单纯预测法则认为只有最末一个历史数据是重要的，未来的预测值即为最末一个历史值。

指数平滑法综合了两种方法，它认为全体历史数据都会影响未来值的预测结果，但全体历史数据并不是同样重要的，时间较近的数据对未来值的影响较大，也较重要；时间较久远的数据对未来值的影响较小，也较不重要。为了区分不同时期历史数据的重要程度，在计算预测值时，指数平滑法为历史数据添加了一个权重。

简单指数平滑是最基本的一种指数平滑方法，它的预测模型由一个预测方程和一个平滑方程构成，其中，预测方程的表达式为 $y_{t+1}=I_t$ ，平滑方程的表达式为 $I_t=\alpha y_t+(1-\alpha)I_{t-1}$ ，显然， I_t 是 α 、 y_t 、 I_{t-1} 的综合表达，其中 I_{t-1} 又可由 α 、 y_{t-1} 、 I_{t-2} 进一步计算得出。

预测方程和平滑方程可写为一个总的指数平滑方程 $y_{t+1}=\alpha y_t+\alpha(1-\alpha)y_{t-1}+\alpha(1-\alpha)^2y_{t-2}+\dots$ ，其中 I_t 被消掉了， α 即为平滑系数，此时可使用 t 个历史数据预测第 $t+1$ 次的值，而在预测第 $t+2$ 次数据时，只需将第 $t+1$ 次的预测值代入指数平滑方程即可。显然，随着历史数据的周期数越来越小，其权重值也越来越小，对预测值的影响也越来越小。

```
> Nile.ses1 <- ses(Nile,h=10,alpha=0.2,intitial="simple")
> Nile.ses2 <- ses(Nile,h=10,alpha=0.6,intitial="simple")
> Nile.ses3 <- ses(Nile,h=10,intitial="simple")
```

程序包 forecast 提供了 ses() 函数用于构建简单指数平滑模型，上述三行代码为 Nile 生成了三个简单指数平滑模型，参数 alpha 指定 Nile.ses1 和 Nile.ses2 的平滑系数分别为 0.2 和 0.6，Nile.ses3 并未指定平滑系数，此时 ses() 函数会自动为其指定一个合适的平滑系数。在三条代码中，都指定参数 intitial 为 simple，即使用前几个观测点的计算结果设置 I_0 。

```
> plot(Nile,xlim=c(1870,1980))
> lines(fitted(Nile.ses1),col="blue",type="o",cex=0.5)
> lines(fitted(Nile.ses2),col="red",type="o",cex=0.5)
> lines(fitted(Nile.ses3),col="green",type="o",cex=0.5)
> lines(Nile.ses1$mean,col="blue")
> lines(Nile.ses2$mean,col="red")
> lines(Nile.ses3$mean,col="green")
```

上述代码可视化地展示了简单指数平滑模型的拟合效果和预测结果，其中，plot() 函数首先绘出了原始数据，参数 xlim 指定横轴的范围为 1870~1980，这为简单指数平滑模型的预测值留下了绘图空间。图 11.14 为 R 返回的绘图空间。

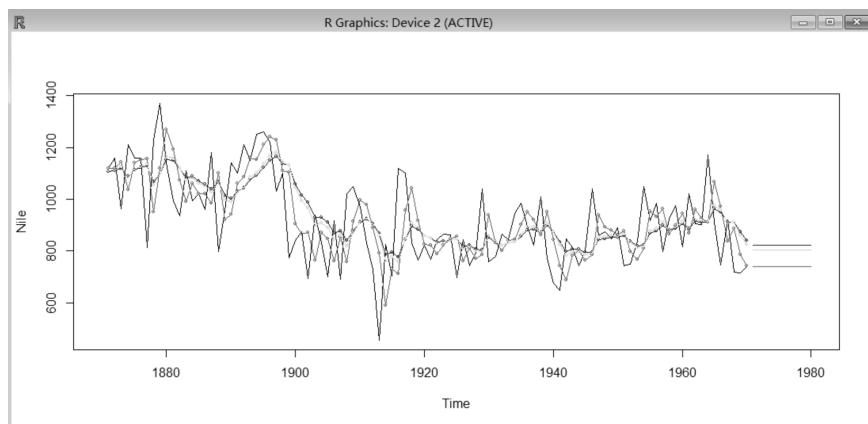


图 11.14 为时间序列 Nile 构建简单指数平滑模型

前三句 `lines()` 函数绘出了三条平滑曲线，`fitted()` 函数计算了模型 Nile.ses1、模型 Nile.ses2 和模型 Nile.ses3 的拟合值，其中 Nile.ses1、Nile.ses2 和 Nile.ses3 的颜色分别为蓝色、红色和绿色，类型 `o` 表示用线将点缀连起来，观察图 11.14 中覆盖在黑色曲线上的三条平滑曲线，显然，平滑系数越大，曲线就越不平滑，平滑系数越小，曲线就越平滑。

后三句 `lines()` 函数则在图形中添加了指平滑模型的预测值，这是三条直线，显然，只有 1972 年的预测值是有意义的，读者也可尝试写一个小程序，使用 1972 年的数据去预测 1973 年，并绘出曲线形式的预测结果。

11.4.3 在指数平滑中加入长期趋势和季节波动

简单指数平滑方法总是假设时间序列中不存在长期趋势和季节波动，Holt 线性趋势方法在简单指数平滑法的基础上添加了长期趋势，它由预测方程、级方程和趋势方程三部分构成，其中预测方程的表达式为 $y_{t+h} = l_t + hb_t$ ，级方程的表达式为 $l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1})$ ，趋势方程的表达式则为 $b_t = \beta*(l_t - l_{t-1}) + (1-\beta*)b_{t-1}$ ，其中， $\beta*$ 为趋势的平滑系数，这三个方程同样可以合为一个方程。

```
> Nile.hlt1 <- holt(Nile,h=10,alpha=0.2,beta=0.2,intitial="simple")
> Nile.hlt2 <- holt(Nile,h=10,alpha=0.8,beta=0.3,intitial="simple")
> Nile.hlt3 <- holt(Nile,h=10,intitial="simple")
```

程序包 `forecast` 提供了 `holt()` 函数用于构建 Holt 线性趋势模型，上述代码同样生成了三个 Holt 线性趋势模型 Nile.hlt1、Nile.hlt2 和 Nile.hlt3。Nile.hlt1 指定指数平滑系数为 0.2，趋势平滑系数为 0.2；Nile.hlt2 则指定指数平滑系数为 0.8，趋势平滑系数为 0.3；Nile.hlt3 并未指定两个平滑系数的值，因此 `holt()` 函数将自动为其生成平滑系数。

```
> plot(Nile,xlim=c(1870,1980))
> lines(fitted(Nile.hlt1),col="blue",type="o",cex=0.5)
> lines(fitted(Nile.hlt2),col="red",type="o",cex=0.5)
> lines(fitted(Nile.hlt3),col="green",type="o",cex=0.5)
> lines(Nile.hlt1$mean,col="blue")
```

```
> lines(Nile.hlt2$mean,col="red")
> lines(Nile.hlt3$mean,col="green")
```

上述 7 条代码同样可视化地表现了 Holt 线性趋势模型的拟合结果和预测结果。观察图 11.15, 红色曲线和蓝色曲线都拟合得有些过头, 其波动幅度比原始数据还要大, 而 Nile.hlt3 对应的绿色曲线则比较好地拟合出了真实数据的波动。在三条预测曲线中, 红色曲线和蓝色曲线都认为 Nile 值会在未来的 10 年内急速下滑, 而绿色曲线则给出了下滑趋势较平缓的 Nile 值预测曲线, 显然, 绿色曲线是最合理的那条曲线。

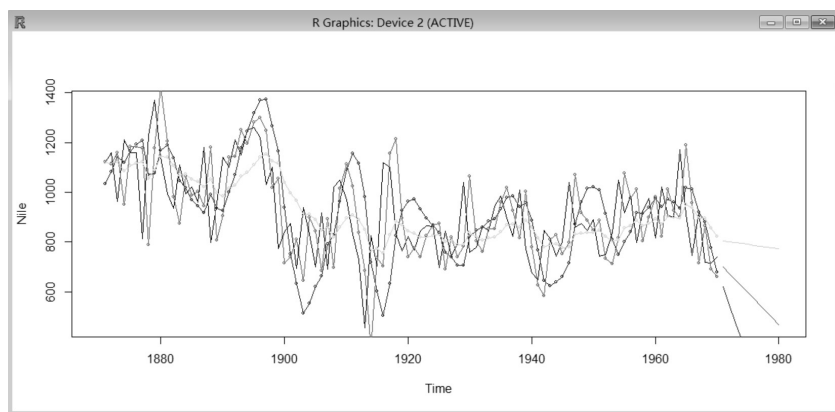


图 11.15 为时间序列 Nile 构建 Holt 线性趋势模型

在 Holt 线性趋势模型中再添加一个季节方程即可在模型中引入季节波动, 此时模型的名称变成 Holt-Winters 季节模型。考虑时间序列的两种分解方法, Holt-Winters 季节模型又分为两种子模型: 一种是加法模型, 另一种是乘法模型。

在加法模型中, 预测方程为 $y_{t+h} = l_t + hb_t + s_{t-m+h}^+$, 级方程为 $l_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$, 趋势方程为 $b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$, 季节方程为 $s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$; 在乘法模型中, 预测方程为 $y_{t+h} = (l_t + hb_t)s_{t-m+h}^+$, 级方程为 $l_t = \alpha \frac{y_t}{s_{t-m}} + (1-\alpha)(l_{t-1} + b_{t-1})$, 趋势方程与加法模型中相同, 季节方程为 $s_t = \gamma \frac{y_t}{l_{t-1} + b_{t-1}} + (1-\gamma)s_{t-m}$ 。其中 $h_m^+ = \lfloor (h-1) \bmod m \rfloor + 1$ 。

```
> AP.hw1 <- hw(AirPassengers,seasonal="additive",h=12)
> AP.hw2 <- hw(AirPassengers,seasonal="multiplicative",h=12)
```

上述代码调用了 hw() 函数用于构建 Holt-Winters 季节模型, 由于 Nile 序列中并不带有季节波动, 因此选择 AirPassengers 数据集作为原始数据, 参数 seasonal 取为 additive 时, 表示使用加法模型预测数据; 参数 seasonal 取为 multiplication 时, 表示使用乘法模型预测数据, hw() 函数中同时指定 h 为 12, 即向后预测 12 个数据, 恰好为一个周期。

```
> plot(AirPassengers,xlim=c(1949,1962),ylim=c(100,700))
> lines(fitted(AP.hw1),col="blue",type="o",cex=0.5)
> lines(fitted(AP.hw2),col="red",type="o",cex=0.5)
> lines(AP.hw1$mean,col="blue")
> lines(AP.hw2$mean,col="red")
```

上述代码同样绘出了 AP.hw1 和 AP.hw2 的拟合曲线和预测曲线，显然，加法模型和乘法模型的拟合曲线都较为紧密地贴合在真实曲线上，但加法模型的预测曲线的波动幅度明显小于乘法模型的预测曲线，乘法模型较好地拟合出曲线随时间增长而剧烈起来的波动。实际上，加法模型适合季节波动较为一致的情况，而乘法模型则适合季节波动随周期的改变按比例变化的情况。

观察图 11.16 的左端，在 1954 年之前，蓝色曲线和红色曲线的拟合结果明显不如 1954 年后的拟合结果好，这与起始阶段历史数据较少、模型能得到的信息不够充分有关。

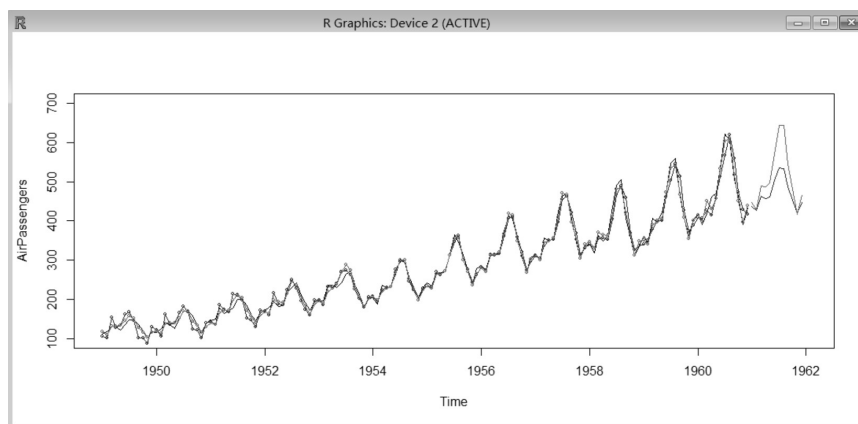


图 11.16 为时间序列 AirPassengers 构建 HW 模型

11.4.4 自回归移动平均模型

自回归移动平均模型是本节最后一种常用的时间序列模型。前面已经介绍过自相关系数的概念，即现在的时间序列与过去的时间序列的相关系数，自回归移动平均模型就是利用时间序列内部的相关关系构造的模型。

自回归移动平均模型可以看作自回归模型和移动平均模型的叠加，自回归模型认为时间序列在时刻 t 的值只和离它最近的 n 个值有关，即 $y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \dots + \alpha_n y_{t-n}$ 。移动平均模型则认为时间序列在时刻 t 的值只和离它最近的 m 个误差有关，即 $y_t = \beta_1 \varepsilon_{t-1} + \beta_2 \varepsilon_{t-2} + \dots + \beta_m \varepsilon_{t-m}$ ，在这里，这 m 个误差指的是在每一时刻进入系统的扰动。

将自回归模型和移动平均模型叠加起来，即可得到自回归移动平均模型，它的公式可归纳为 $y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \dots + \alpha_n y_{t-n} + \beta_1 \varepsilon_{t-1} + \beta_2 \varepsilon_{t-2} + \dots + \beta_m \varepsilon_{t-m}$ ，这显然是一个回归模型，对自回归移动平均模型的求解也就转为对 n 个 α 系数和 m 个 β 系数的求解。

最后一个和自回归移动平均模型有关的概念是差分，一阶差分可写为 $y'_t = y_t - y_{t-1}$ ，二阶差分则为 $y''_t = y'_t - y'_{t-1}$ ，在模型中引入差分的概念可以消除时间序列中的不平稳性。程序包 forecast 提供了 auto.arima() 函数用于构建自回归移动平均模型，它能够自动确定出最佳的回归系数。

```
> auto.arima(Nile)
Series: Nile
ARIMA(1,1,1) with drift
```

```

Coefficients:
      ar1      ma1      drift
      0.2707 -0.9054 -2.8827
s.e.  0.1182  0.0579  2.0270
sigma^2 estimated as 19604:  log likelihood=-623.46
AIC=1254.91   AICc=1255.34   BIC=1265.29

```

上述代码为时间序列 Nile 构建了自回归移动平均模型，R 返回的结果显示这是一个带漂移的 ARIMA(1,1,1) 模型，模型中的三个 1 按顺序分别表示模型中自回归的阶、差分级数和移动平均的阶数分别为 1。根据 Coefficients 元素给出的系数，可得模型方程为 $y'_t = 0.2707 y'_{t-1} + \varepsilon_t - 0.9054 \varepsilon_{t-1} + c$ ，其中 $y'_t = y_t - y_{t-1}$ ， c 为误差项。

在上述模型中， ε_t 与 c 都是取值为随机分布的白噪声，因此实际上不可能为模型方程计算出一个精确的值来。R 同样返回了模型的方差，其值为 19 604，是一个非常大的数，说明模型的预测值在一个较大的范围内波动。

```

> Nile.au <- auto.arima(Nile)
> Nile.fr <- forecast(Nile.au)
> plot(Nile.fr)

```

上述代码使用 forecast() 函数计算了 ARIMA 模型的预测值，并使用 plot() 函数将其绘制了出来，图 11.17 是 R 的返回结果。观察图 11.17，在真实数据后边添加的 10 个预测值呈现略向下倾斜的趋势，这是漂移系数对模型造成的影响，这些预测值同样有较大的置信区间，这也与模型的方差较大相吻合。

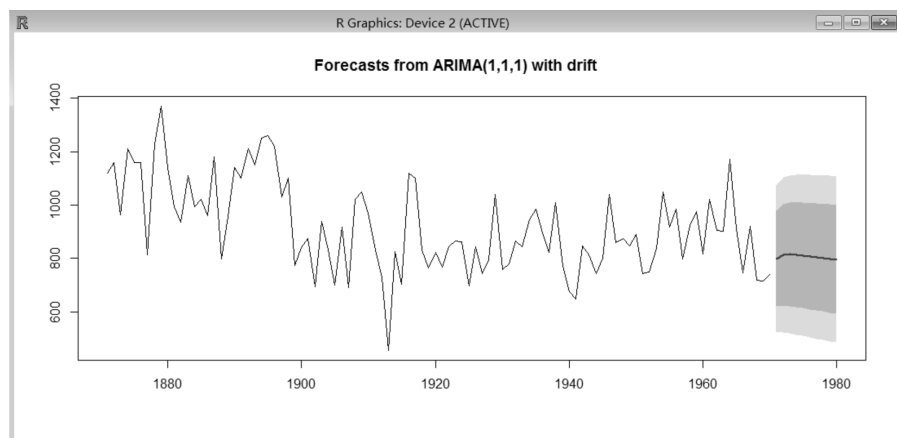


图 11.17 为时间序列 Nile 构建 ARIMA 模型

```

> auto.arima(AirPassengers)
Series: AirPassengers
ARIMA(0,1,1) (0,1,0) [12]

Coefficients:
      ma1

```

```

-0.3184
s.e.    0.0877

sigma^2 estimated as 137.3:  log likelihood=-508.32
AIC=1020.64  AICc=1020.73  BIC=1026.39

```

上述代码对 `AirPassengers` 构建了 ARIMA 模型，`AirPassengers` 是一个带有明显的季节趋势的时间序列，`auto.arima()` 函数自动在 ARIMA 模型中添加了季节趋势，R 返回的模型为 `ARIMA(0,1,1)(0,1,0)[12]`，其中第一组符号 (0, 1, 1) 表明模型的自回归阶数、差分阶数、移动平均阶数分别为 0、1、1，第二组符号 (0, 1, 0) 表示该序列中季节波动的自回归阶数、差分阶数、移动平均阶数分别为 0、1、0，第三个符号 [12] 则表明该序列以 12 个数据为一个周期。

与非季节性的 ARIMA 模型相比，季节性的 ARIMA 模型在方程中增加了有关季节的回归项。差分形式不需要系数，因此上述代码仅给出了一阶移动平均阶数的系数，该模型的方差为 137.3，处于一个较为合理的范围内。

```

> AP.au <- auto.arima(AirPassengers)
> AP.fr <- forecast(AP.au)
> plot(AP.fr)

```

上述代码使用 `forecast` 函数拟合了时间序列 `AirPassengers` 在 ARIMA 模型下的预测值，并将其绘制了出来，如图 11.18 所示，ARIMA 模型为 `AirPassengers` 向后预测了两个周期，其波动形状与时间序列的波动趋势基本吻合，图 11.18 同样添加了置信区间，与图 11.17 相比，图 11.18 中的置信区间明显要小，即预测结果更可信。此外，随着预测周期数的增加，置信区间的浮动也剧烈了起来，这说明较近的预测结果要比较远的预测结果可靠。

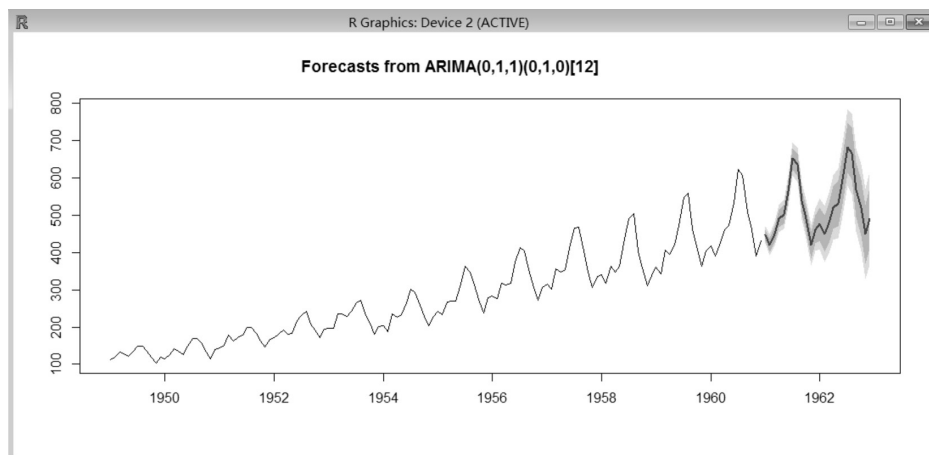


图 11.18 为时间序列 `AirPassengers` 构建 ARIMA 模型

第 12 章 R 中的最优化问题

本章将深入浅出地介绍有关最优化的知识，包括黄金分割法、牛顿法和最快上升法等，并讨论 R 自带的最优化函数。本章涉及大量的高等数学知识，通过阅读本章，读者将初步了解最优化的应用和基础方法，并将更深入地理解前文中已介绍的统计方法。

12.1 最优化问题简述

到目前为止，我们已经接触了许多有关最优化的问题，比如在计算回归方程中的回归系数时 R 会选择最优的系数，再比如选择自回归阶数、差分级数和移动平均阶数使 ARIMA（自回归积分滑动平均模型）达到最优，等等。

现实生活中，许多问题都可变形、简化为最优化问题，掌握最优化方法有助于解决某些复杂的问题，比如当我们想使用多项式回归来拟合自变量和因变量时，我们希望选择最好的多项式阶数，既不会太小使得模型拟合结果过差，也不会太大使得模型出现过拟合的情况，对新数据的预测十分差。

在已知的最优化问题中，有一些问题的最优化方法是清楚的，比如在逐步回归中我们知道要使用指标 AIC 判断模型的优劣，也知道指标 AIC 的计算方法；而在另一些问题中，模型的最优化则是一个黑盒，比如我们仅知道 `auto.arima()` 函数输出了一个结果，而不知道这个结果是怎样计算出来的。

到目前为止，计算机仍不能像人类一样进行思考，即在寻找一个方程的最优解时，计算机难以像人类一样将方程简化变形，并求出最终结果，计算机擅长将所有可能的解一个一个地代入方程中，并找出使方程达到最大值或最小值的那个解，显然，由于方程的解分布在全体实数集上，从这无穷多个解中找出最佳的解是一件比较困难的事情，因此有时我们会转而关心在一定范围内最佳的那个解，并称在一个小范围内使函数达到最值的解为局部最优解。

在局部最优解的求解过程中有一些通用的准则： $|x(n)-x(n-1)| \leq \varepsilon$ ， $|f(x(n)) - f(x(n-1))| \leq \varepsilon$ ， $|f'(x(n))| \leq \varepsilon$ 。假设 x 为局部最优解， $x(n)$ 则为分散在 x 附近的一些点，并且随着 n 的增大， $x(n)$ 将无限接近于 x ，此时上述三个条件将成立。

在求解局部最优解时，计算机在 x 附近自动生成一些序列，考虑到并非所有满足上述条件的 x 都是局部最优解，当在 x 附近找到一个同时满足上述三个条件的序列时，仍需进一步检验 x 是否真的是局部最优解。

本章讨论的重点是最优化方法，即如何求出一个函数的全局最优解或局部最优解，而不涉及如何将实际问题抽象为最优化问题，即如何找到恰当的最优化函数来表达实际问题。本章介绍了黄金分割法、牛顿法和最快上升法等最优化方法，它们涉及较高深的

高数知识。本章尽可能深入浅出地讲解理论部分，但仍假设读者有基本的高等数学方面知识。

此外，本章仅涉及了最一般化的情况，即连续变量的无约束最优化问题，对于离散变量最优化、或有约束的最优化来说，黄金分割法、牛顿法和最快上升法将不起作用。其他一些著名的最优化方法包括抛物线差插补法、内尔德—米德算法、共轭梯度法和模拟退火法等，这些算法在机器学习领域格外重要，也可以移植到其他语言中。

12.2 黄金分割法

黄金分割法是较为简单的一种最优化方法，本节将讨论黄金分割法的背景思想和实现方法，并用它实际地处理一个函数求最值的问题。

12.2.1 黄金分割法和局部最优解

黄金分割法是一种只能应用于一维情况的最优化方法，即仅关心一个自变量和因变量之间的关系。考虑求解最大值的情况，如果在自变量区间 $[x, z]$ 中，存在一点 y 使得 $f(y)$ 同时大于 $f(x)$ 和 $f(z)$ ，那么在区间 $[x, z]$ 中一定存在一个点，在该点处的一阶导数为 0，二阶导数小于 0，这个点可能不是 s ，也可能就是 s 。显然，这就是一个局部最大值。

黄金分割法通过不断将区间切割为更小的区间以逼近最大值，当区间的左、右端点极为近似时，我们也就近似地估计出了局部最大值的解。在切割区间时，我们希望能尽快地把区间切割到足够小，同时还希望切割出尽可能多的区间，尽量多比较一些区间端点。

黄金分割法使用黄金分割值作为切割比例点，黄金分割值指 $\frac{1+\sqrt{5}}{2}$ ，在图 12.1 中， $\frac{b}{a}$ 和 $\frac{a}{c}$ 的值都为黄金分割值，以 a 为宽、以 b 为长的大矩形能切为一个以 a 为边长的正方形和一个以 c 为宽、以 a 为长的小矩形，此时小矩形和大矩形满足相似关系。

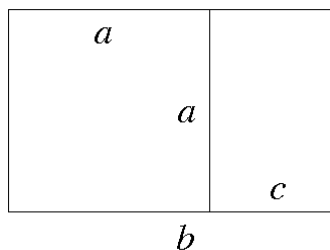


图 12.1 黄金分割示意图

数学家已经证明，在逼近的求解局部最优值时，使用黄金分割值分割区间，函数能得到最好的收敛效果，即同时顾虑到收敛速度和结果的准确度，显然，这是一个很神奇的比值。

图 12.2 给出了黄金分割法一次迭代的示意图，在分割法开始工作之前，首先要确定出分割法工作的区间范围，即图中的 x 点和 z 点，以及一个函数值大于 x 点和 z 点的点，

即图中的 s 点。此时区间 $[x, z]$ 已被切割为区间 $[x, s]$ 和区间 $[s, z]$ 。

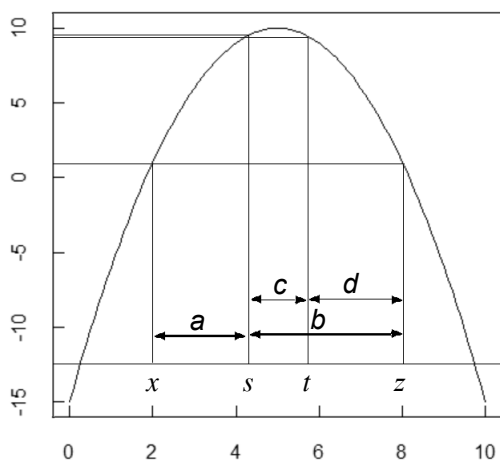


图 12.2 黄金分割法示意图

我们希望将其夹逼为更小的区间，首先比较区间 $[x, s]$ 和区间 $[s, z]$ 的长度大小，并在较长的那个区间中取一个切割值，图 12.2 中的区间 $[x, s]$ 的长度小于区间 $[s, z]$ ，因此在区间 $[s, z]$ 中取了一个新的点 t ，点 t 将区间 $[s, z]$ 进一步切分为区间 $[s, t]$ 和区间 $[t, z]$ 。

点 t 的选择并不是任意的，记区间 $[x, s]$ 、 $[s, z]$ 、 $[s, t]$ 、 $[t, z]$ 的长度分别为 a 、 b 、 c 、 d ，那么 t 点的选择需满足 $\frac{b}{a} = \frac{d}{c}$ ，以及 $\frac{a}{c} = \frac{b}{d}$ 。

确定好 t 的值后，比较点 t 和点 s 对应函数值的大小，如果 $f(t)$ 大于 $f(s)$ ，则将点 x 更新为点 s ，点 s 更新为点 t ，此时由于 $f(s)$ 大于 $f(z)$ ，故 $f(t)$ 也大于 $f(z)$ ，满足黄金分割法的假设，因此认为局部最大值位于区间 $[s, z]$ 上。如果 $f(t)$ 小于 $f(s)$ ，则将点 z 更新为点 t ，此时 $f(s)$ 仍为 $f(s)$ 、 $f(x)$ 、 $f(t)$ 中的最大值，局部最大值位于区间 $[x, t]$ 上。

显然，无论新区间是 $[s, z]$ 还是 $[x, t]$ ，都使得新区间中较大区间和较小区间的长度之比与旧区间 $[x, z]$ 中相同。记黄金分割值 $\frac{1+\sqrt{5}}{2}$ 为 q ，则可得方程式 $t = s + \frac{z-s}{1+q}$ 。类似地，当区间 $[x, s]$ 的长度大于区间 $[s, z]$ 时，也可在区间 $[x, s]$ 中取得点 t ，其计算公式为 $t = s - \frac{z-s}{1+q}$ 。

12.2.2 使用 R 实现黄金分割法

理解了黄金分割法在一次迭代中如何判断 t 的取值，并对区间进行夹逼后，很容易将其推广到多次迭代的情形中，当迭代次数足够多后，含有局部极大值的区间将被夹逼得足够小，此时可以用区间中的任意一个点近似代替局部最大值。

```
> section <- function(f,x,z,s,stp=1e-5){
+   q <- 1+(1+sqrt(5))/2;fx <- f(x);fz <- f(z);fs <- f(s)
+   while((z-x)>stp){
+     if((z-s)>(s-x)){
```

```

+   t <- s+(z-s)/q;ft <- f(t)
+   if(ft>=fs){x <- s;s <- t;fx <- fs;fs <- ft}
+   else{z <- t;fz <- ft}
+ }
+ else{
+   t <- s-(z-s)/q;ft <- f(t)
+   if(ft>=fs){z <- s;s <- t;fz <- fs;fs <- ft}
+   else{x <- t;fx <- ft}
+ }
+ points(t,ft,col="red")
+ }
+ return(t)
+ }

```

上述代码创建了一个函数 `section`，它接受 5 个参数，其中，参数 f 准备传入被计算的函数， x 、 z 、 s 则是 x 点、 z 点、 s 点的初始值，`stp` 给出了一个非常小的数 0.000 05，它用于判断区间是否已夹逼近到足够小的程度。

在函数体中，第 1 行代码首先创建了 4 个变量，其中 p 为黄金分割值， fx 、 fz 、 fs 则分别为 x 点、 z 点、 s 点对应的函数值。从第 2~14 行代码是一个 `while` 循环，`while` 循环的循环条件为 $(z-x)>stp$ 为真，即只要区间 $[x, z]$ 未夹逼近到比 0.000 05 小的程度，`while` 循环就一直运行下去。

`while` 循环的循环体主要由一组 `ifelse` 语句组成，该语句的两条分支下又有一组 `ifelse` 语句。当区间 $[s, z]$ 的长度大于区间 $[x, s]$ 时，`ifelse` 语句进入第一个分支，此时创建了变量 t ，并再次使用 `ifelse` 语句判断 ft 与 fs 的大小关系，并根据判断结果更新了区间。当区间 $[s, z]$ 的长度小于等于区间 $[x, s]$ 时，`ifelse` 语句进入第二个分支，此时使用另一个公式创建了 t ，并再次判断 ft 与 fs 的大小关系，更新了区间。

在 `while` 循环中的最后一句代码使用 `points()` 函数在图形中添加了以 t 为横坐标、以 ft 为纵坐标的点，显然，区间每迭代一次，图形中就会添加一个新的点。`return()` 语句返回了最后一次迭代中 t 的值，由于此时区间 $[x, z]$ 已经非常小，因此可以将 t 值近似视为局部最小值。

```

> fun <- function(x){-(x-5)^2+25}
> a <- seq(0,10,0.01)
> b <- -(a-5)^2+25
> plot(a,b,type="l")
> section(f=fun,x=0,z=9,s=2)
[1] 4.999998

```

构建好 `section` 函数后，不妨在一个简单函数上实际检验一下它的效果。上述代码首先构建了一个二次函数 `fun`，其表达式为 $y=-(x-5)^2+25$ ，显然，这个二次函数仅在坐标 (5, 25) 处有一个最大值。在函数 `section` 中我们预先写下了 `points()` 函数，因此还需构建一个基础图形以供函数 `section` 向上添加点。在上述代码中，变量 a 是从 0 到 10、间隔为 0.01 的序列，变量 b 则是 a 对应的函数值，利用 `plot()` 函数为变量 a 和变量 b 绘出了一张曲线图。

准备好函数和基础图形后，上述代码调用了函数 `section`，在最后一行命令中，指定参数 f 为 `fun`， x 、 z 、 s 分别为 0、9、2，由于 2 对应的函数值大于 0 和 9 对应的函数值，因此这一组参数是符合黄金切割法的假设的。R 返回的值为 4.999 98，这是一个非常贴近正确解的值，函数 `section` 显然执行了它的功能。R 同样返回了一个图形，其返回的图形如图 12.3 所示。

观察图 12.3，图中绘出了一条光滑的曲线，并在曲线上绘出了一些红色的小圆圈，这些小圆圈对应的横坐标在区间 $[2, 8]$ 之间来回跳跃，并在最后稳定到 5 附近，显然，它们反映了点 t 的波动情况。函数 `section` 中预留了绘制小圆圈的 `points` 语句，如果不想要绘制图形，则除去掉 `plot()` 函数外，还需在函数 `section` 中将 `points` 语句去掉。

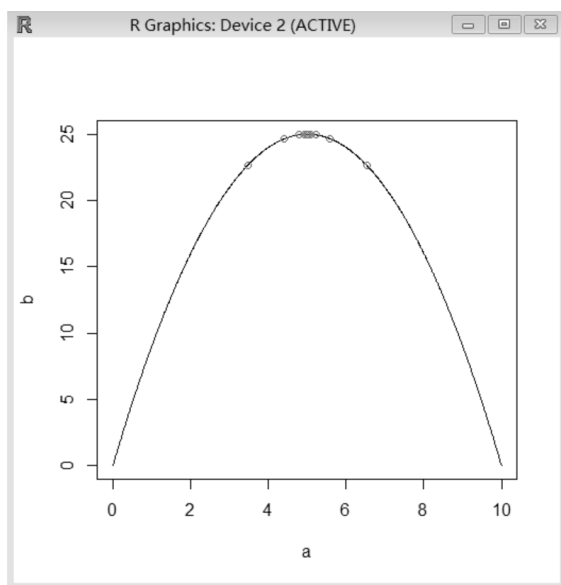


图 12.3 绘制二次函数和 t 值的变化曲线

图 12.3 中仅有一个最大值，黄金切割法较为容易地迭代到了正确解附近，对于包含多个局部最大值的函数来说，迭代也许要进行很多次，比如函数 $\sin(x)$ 在 $[0, 10]$ 上有两个最大值，分别是 $\frac{\pi}{2}$ 和 $\frac{5\pi}{2}$ ，如果令函数 `section` 对这个函数求解，函数 `section` 将艰难地迭代好长时间，才会将区间夹逼到某一个解上去，具体是哪个解则由初始值决定。

此外，本节仅讨论了求解最大值的情况，对于求解最小值的问题来说，只需在函数前加一个符号即可变为求解最大值的问题。

12.3 牛顿最优化方法

牛顿法是一种既能应用于一维情况，也能应用于多维情况的最优化方法。与黄金分割法相比，牛顿法具有更简洁的形式，本节将讨论牛顿法的思想和应用，并使用 R 检验其效果。

12.3.1 牛顿法的算法原理

黄金分割法仅应用了被求解函数本身的信息，牛顿法则考虑在求解局部最优解时使用导数信息。在前面已经提到过，如果一个点是局部最优解，那么它的一阶导数必然是0，那么求解局部最优解的问题就转化为求解一阶导数为0的点。

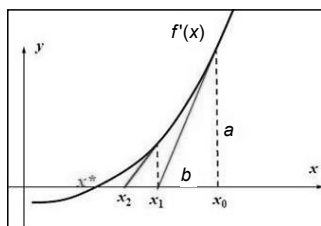


图 12.4 牛顿迭代法示意图

在一维情况下，牛顿-拉弗森算法给出了迭代地逼近局部最优解的公式：
$$x(n+1) = x(n) - \frac{f'(x(n))}{f''(x(n))}$$
图 12.4 直观地给出了该算法工作的原理。在图 12.4 中，绘出的曲线为 $f'(x)$ ，我们希望求出该曲线与横坐标相交的点。不妨设从 x_0 点开始迭代， $f'(x_0)$ 对应的长度为 a ， $f''(x_0)$ 对应的是 x_0 点的斜率，即 $\frac{a}{b}$ ，令 $f'(x_0)$ 与 $f''(x_0)$ 做商，求得的结果为 b ， x_0 减去 b 后便迭代到了 x_1 的位置，再次迭代， x_1 转移到了 x_2 的位置，显然，最终 x_n 会无限逼近 x^* ，即 $f(x)$ 的局部最优解。

从一维情形推广到多维时，牛顿迭代法变得复杂了许多。对于同一点 x_n ，在一维情形下只需求解它的一阶导数和二阶导数，而在多维情形下，则需求解它的几个偏导数，偏导数的个数由函数维度决定。为了便于讨论，我们首先给出几个定义。

不妨设函数中一共有 d 个自变量，分别记为 x_1, x_2, \dots, x_d ，则第 i 个偏导数记为 $f_i(x) = \frac{\partial f(x)}{\partial x_i}$ ，其中 x 为 d 个自变量组成的向量， ∂ 为求偏导符号，对 x_i 求偏导时将其他 $d-1$ 个自变量看作常数，再对 x_i 求导。偏导数的定义还可引申出梯度的定义：

$$\nabla f(x) = (f_1(x), \dots, f_d(x)), \text{ 黑塞矩阵则定义为: } H(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1 \partial x_1} & \dots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_d \partial x_1} & \dots & \frac{\partial^2 f(x)}{\partial x_d \partial x_d} \end{bmatrix}, \text{ 此时有牛顿}$$

迭代公式：
$$x(n+1) = x(n) - H(x(n))^{-1} \nabla f(x(n)).$$

类比一维情形下的导数，设定 $x(0)$ 后，多维情形下的牛顿迭代法公式也会收敛到一个梯度分量全为 0 的点，在这个点附近不存在任何一个比该点还“低”或者还“高”的点，这个点就是函数的一个局部最优解，也就是我们所要寻找的点。

在三维情形下可以直观理解梯度的几何意义，不妨想象求解的过程是一个登山的过程，在山腰处向山顶出发时，我们要上山，此时梯度是一个正数；在山腰处向山脚出发时，

我们要下山，此时梯度是一个负数。显然，在山顶处找不到任何比它还高的点，此处的梯度处处为 0；山脚处找不到任何比它还低的点，此处梯度也处处为 0。

值得注意的是，最小值和最大值的一阶导数值都为 0，因此牛顿法并不确定求出的是最小值还是最大值，不同的初始值会导致不同的结果。在实际应用中，我们总需要对牛顿法返回的结果进行判断。此外，牛顿法假设函数的一阶导数和二阶导数都是连续的，如果函数不满足该假设，显然，黑塞矩阵将在不连续点返回默认值，牛顿法就无法正常工作。

12.3.2 在一维情形下实现牛顿迭代法

在 R 中实现牛顿迭代法时同样需要我们编写一个新函数，与黄金分割法类似，我们考虑在 $x(n)$ 的一次导数小于 0.000 01 时即近似地认为 $x(n)$ 的一次导数就是 0， $x(n)$ 就是局部最优解。

但与黄金分割法不同的是，黄金分割法中已经指定了算法运行的区间，初始区间总能被切割到一个足够小的范围，而牛顿迭代法并不指定初始区间，因此，对于不存在局部最优解的函数来说，牛顿迭代法会无限循环下去，因此还需指定最大的迭代次数，当超过这个次数后仍未迭代出正确结果时，就停止迭代。

```
> newton1 <- function(f,f1,f2,x,stp=1e-5,stn=100){
+   fx <- f(x);f1x <- f1(x);f2x <- f2(x);n <- 0
+   while((abs(f1x)>stp)&(n<stn)){
+     x <- x-f1x/f2x
+     fx <- f(x);f1x <- f1(x);f2x <- f2(x)
+     n <- n+1
+     points(x,fx,col="red")
+   }
+   if(n==stn){print("can't find a right result")}
+   else{return(x)}
+ }
```

上述代码创建了一个函数 `newton1`，实现了在一维情形下执行牛顿迭代法的功能。函数 `newton1` 接受 6 个传入参数，参数 f 、 $f1$ 、 $f2$ 分别接受被求解函数的原函数、一次导函数和二次导函数。参数 x 用于指定 $x(n)$ 的初始值，牛顿迭代法将从 x 点处开始迭代。

参数 `stp` 为默认的 $x(n)$ 与局部最优解的最大误差，参数 `stn` 为默认的最大迭代次数，这两个参数用于给出函数停止工作的条件。在上述代码中，这两个参数的默认值为 0.000 01 和 100，`stp` 越小、`stn` 越大，牛顿迭代法给出的结果精度就越高。

在函数体中，第 1 行代码首先计算了 x 的函数值、一次导数值和二次导数值，并分别赋给了变量 fx 、 $f1x$ 、 $f2x$ ， n 为迭代次数，它的初始值为 0。`while()` 函数创建了一个循环，该循环有两个循环条件， $f1x$ 的绝对值大于 0.000 01 及迭代次数 n 小于 100，符号 `&` 表示这两个条件是并列关系，只有这两个条件都满足时，循环体才会执行。

循环体中给出了迭代过程，它首先利用牛顿迭代法的公式更新了 x 的值，然后计算了新的 fx 、 $f1x$ 、 $f2x$ ，并将迭代次数增加了 1，`points()` 函数将点 (x,fx) 添加到了一个图形

中，函数 `newton1` 并未画出这个图形，因此，在调用 `newton1` 时需要预先绘出一张图形。

循环体结束后，`ifelse` 语句判断循环体的结束条件，如果循环体的循环次数与最大循环次数相同，那么此时牛顿迭代法并没有迭代出正确结果，因此返回一个提示语句；如果循环体是由于 $f1x$ 的绝对值小于 0.000 01 而停止工作，那么此时 x 就是局部最优解，返回 x 的值。

```
> fun <- function(x){-(x-5)^2+25}
> fun1 <- function(x){-2*x+10}
> fun2 <- function(x){-2}
> a <- seq(0,10,0.01)
> b <- -(a-5)^2+25
> plot(a,b,type="l")
> newton1(f=fun,f1=fun1,f2=fun2,x=8)
[1] 5
```

准备好牛顿迭代函数后，上述代码创建了三个函数 `fun`、`fun1` 和 `fun2`，其中函数 `fun` 的表达式为 $y=-(x-5)^2+25$ ，函数 `fun1` 的表达式为 $y=-2x+10$ ，函数 `fun2` 的表达式为 $y=-2$ ，显然 `fun1` 是 `fun` 的一次导函数，`fun2` 是 `fun` 的二次导函数。

第 3~6 行代码绘出了一张基础图形，最后一行代码则调用了函数 `newton1`，将 `fun`、`fun1`、`fun2` 作为原函数、一次导函数和二次导函数传入了函数 `newton1`，并设置 x 的初始值为 8。函数 `newton1` 返回了数值 5，并在基础图形中添加了点。

观察图 12.5，数值 5 确实是原函数 `fun` 的一个最值，并且图中仅有一个红色的小圆圈，这表示函数 `newton1` 仅迭代了一次便求出了最终结果。考虑一次导函数的形状，它是一条直线，斜率处处相同，且和横坐标只有一个交点，因此无论 $x(n)$ 的初始值是多少， $x(n)$ 都会收敛到 5，并且迭代次数总是为 1。

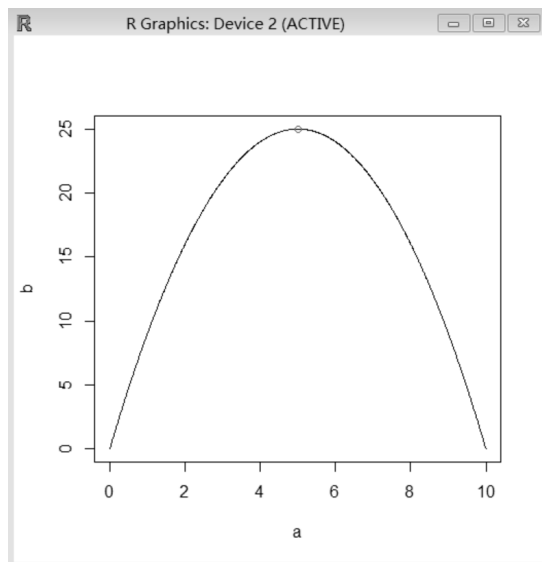


图 12.5 绘制原函数和 x 点

函数 `newton1` 和函数 `section` 对相同的函数求解了局部最优值，显然，`newton1` 的迭代次数远远少于 `section`，实际上，牛顿迭代法的速度确实优于黄金分割法，即便在多维情形中，牛顿迭代法也是一种相当快的最优化方法。

12.3.3 在多维情形下实现牛顿迭代法

将牛顿迭代法从一维情形推展到多维情形时，需要在函数 `newton1` 的基础上进行修改。在 `newton1` 中仅用到了 f 、 f_1 、 f_2 这三个函数，在多维情形中，由于我们并不确定被处理的原函数有多少个自变量，因此需要考虑到求解局部最优解的函数的兼容性。

```
> newton2 <- function(f,x,stp=1e-5,ntp=100){
+   fx <- f(x);n <- 0
+   while((max(abs(fx[[2]]))>stp)&(n<ntp)){
+     x <- x-solve(fx[[1]],fx[[2]])
+     fx <- f(x);n <- n+1
+   }
+   if(n==ntp){print("can't find a right result")}
+   else{return(x)}
+ }
```

上述代码创建了函数 `newton2`，它接受 4 个传入参数，其中参数 f 应当是一个包含黑塞矩阵和梯度向量的列表，由于并不确定被求解局部最优解的函数有几个自变量，因此也无法在 `newton2` 中写入计算黑塞矩阵和梯度向量的语句，将计算黑塞矩阵和梯度向量的语句放在调用函数 `newton2` 之前执行。参数 x 则是初始值的向量坐标，参数 `stp` 是 $x(n)$ 和最优解的最大误差，参数 `ntp` 则是最大迭代次数。

在函数体中首先创建了变量 fx 的，并计算了它值，假设 fx 是一个包含两个元素的列表，第一个元素是黑塞矩阵，第二个元素是梯度向量，并创建了代表迭代次数的变量 n ，`while` 循环实现了求出最优解的功能。

`while` 循环同样有两个循环条件，第一个循环条件用于判断 x 的梯度向量中绝对值最大的分量是否大于 `stp`，由于最优解处的梯度向量每一个分量都为 0，故我们希望最优解的近似解中每一个分量也近似为 0，其中绝对值最大的分量也不能超过误差值。第二个循环条件则与 `newton1` 中一样，判断迭代次数是否超过最大值。

在循环体中，`solve()` 函数求出了黑塞矩阵的逆和梯度向量的乘积，即 $H(x(n))^{-1}\nabla f(x(n))$ 部分，注意，`solve()` 函数中的参数不能随意调换位置。更新了 x 的值后，循环体也更新了 fx 的值和迭代次数。在循环结束后，`newton2` 同样使用 `ifelse` 语句判断了循环体结束的条件，并根据判断结果返回了一条提示信息或最优解的值。

```
> fun <- function(x){
+   fun0 <- x[1]^2/2+x[2]^2/4
+   fun1 <- x[1]+x[2]^2/4
+   fun2 <- x[1]^2/2+x[2]/2
+   fun11 <- 1+x[2]^2/4
+   fun12 <- x[1]+x[2]/2
+   fun22 <- x[1]^2/2+1/2
+ }
```

```
+ return(list(matrix(c(fun11, fun12, fun12, fun22), 2, 2), c(fun1, fun2)))
+ }
```

上述代码构建了函数 `fun`，它接受一个参数 x ，它和函数 `newton2` 中的参数 x 相同，是一个给出了初始值坐标的向量。在函数 `fun` 中，`fun0` 计算了原函数的值，原函数的形式为 $\frac{x_1^2}{2} + \frac{x_2^2}{4}$ ；`fun1` 求解了 x_1 的偏导数，其计算公式为 $x_1 + \frac{x_2^2}{4}$ ；`fun2` 求解了 x_2 的偏导数，其计算公式为 $\frac{x_1^2}{2} + \frac{x_2}{2}$ ；`fun11` 对 x_1 求了两次偏导数，`fun12` 则对 x_1, x_2 分别求了一次偏导数，`fun22` 则对 x_2 求了两次偏导数。

`return` 语句返回了函数 `newton2` 所需要的列表元素，它包含两个元素，其中第一个元素是根据原函数计算得到的黑塞矩阵，其具体形式为
$$\begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2 \partial x_2} \end{bmatrix}$$
，改变自变量求偏导的顺序并不影响最终结果，因此 $\frac{\partial^2 f(x)}{\partial x_1 \partial x_2}$ 和 $\frac{\partial^2 f(x)}{\partial x_2 \partial x_1}$ 均由 `fun12` 给出。

```
> newton2(f=fun, x=c(1, 1))
[1] -1.345347e-08 -8.027479e-09
```

上述代码调用了函数 `newton2`，并将参数 f 指定为函数 `fun`，将初始值设定为 $(1, 1)$ ，该初始值最终迭代到了 $(0, 0)$ 的位置，这显然是原函数的最低点，它对应的函数值为 0，其他点对应的函数值都将大于 0。在这个例子中，修改初始值的坐标将得到不同的局部最优解。

12.4 最快上升法

本节将讨论最后一种常用的最优化方法，与牛顿迭代法相比，最快上升法使用了更多与梯度计算有关的公式，其思想与其他常用的最优化方法也更加相似。

12.4.1 利用梯度求解上升最快的相邻点

在牛顿迭代法中我们已经了解了偏导数的定义，并引申出了梯度的概念和黑塞矩阵的计算方法。在多维情况下，牛顿迭代法需要用到一个很大的黑塞矩阵，而黑塞矩阵又需要我们手动计算得出，这是很不方便的。最快上升法抛掉了黑塞矩阵的部分，转而尝试使用梯度求解局部最优解。

假设 v 是一个 d 维的非零向量，那么 v 的范数就是 $\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_d^2}$ ，函数在 x 点处沿着 v 方向的斜率可定义为 $\frac{v^T \nabla f(x)}{\|v\|}$ ，显然，斜率为正时，对应的方向在上升，

斜率为负时, 对应的方向在下降。假设取一个序列, 使其迭代地逼近局部最优解, 那么相邻的两个点总可以写为 $x(n+1)=x(n)+\alpha v$ 的形式, 其中 α 是一个不小于 0 的数。在每一次迭代中, 我们总希望找到合适的 α 、 v , 使得在 $x(n)$ 固定时, $x(n+1)$ 对应的函数值能取到最大, 即 $x(n)$ 点能上升到附近最高的一个点处。

令斜率对 v 中的每一个分量求偏导, 第 i 个分量对应的结果为 $\frac{f_i(x)}{\|v\|} - \frac{(v^T \nabla f(x)) v_i}{\|v\|^3}$,

显然, 梯度 $\nabla f(x)$ 时 x 点处斜率最大的方向, 此时 $x(n+1)$ 又可写为 $x(n+1)=x(n)+\alpha \nabla f(x(n))$, 最快上升法所要求解的即为 α 的具体值, 该值应使 $x(n+1)$ 对应的函数值达到最大。

显然, α 是一个数值, 求解 α 的最优解即为一维空间中的最优解问题, 黄金分割法在这里能够起到帮助作用。

```
> fit.alpha <- function(f,x,grad.fx,stp=1e-5){
+   if(sum(abs(grad.fx)==0)){return(x)}
+   else{
+     g <- function(alpha){f(x+alpha*grad.fx)}
+     xx <- 0;gx <- g(xx);xs <- 1;gs <- g(xs)
+     while((gs<gx)&(xs>stp)){xs <- xs/2;gs <- g(xs)}
+     if(xs<=stp){return(x)}
+     else{
+       xz <- 2*xs;gz<-g(xz); n <- 0
+       while((gs<gz)&(n<100)){xs <- xz;gs <- gz;xz <- 2*xs;gz <- g(xz);n
+       <- n+1}
+       if(xs>=100){return(x+xs*grad.fx)}
+       else{
+         alpha <- section(g,xx,xz,xs)
+         return(x+alpha*grad.fx)
+       }
+     }
+   }
+ }
```

上述代码创建了一个函数 `fit.alpha`, 它用于求解 α 的具体值, 函数 `fit.alpha` 接受 4 个参数, 其中参数 f 为原函数, 参数 x 为初始值, 参数 `grad.fx` 为 x 点的梯度, 参数 `stp` 则是一个误差项。

函数体中首先使用了一组 `ifelse` 语句用于判断 x 点的梯度向量是否所有的分量均为 0, 如果 `if` 语句的判断结果为真, 则说明 x 点就是一个最优解, 此时返回 x ; 否则进入 `else` 语句块, 在 `else` 语句块中, 首先创建了一个函数 g , 它接受参数 α , 并返回 $f(x+\alpha \nabla f(x))$, 即 $x+\alpha \nabla f(x)$ 。

我们希望使用黄金分割法确定最佳的 α 值, 故首先需要找出一组 x 、 s 、 z , 使 s 位于区间 $[x, z]$ 上, 并满足 $f(s)$ 同时大于 $f(x)$ 和 $f(z)$, 考虑到 α 是一个不小于 0 的数, 故不妨令 x 点为 0, 为了和参数 x 进行区分, 将区间的左端点命名为 xx , gx 是 xx 点对应的原函数值。令 xs 代表 s 点, 不妨首先令它取为 1, gs 是 xs 点对应的原函数值。

创建好变量后, 初始的 s 点未必满足 $f(s)$ 大于 $f(x)$ 的假设条件, `while` 循环给出了两个循环条件, 第一个条件为 $f(s)$ 小于 $f(x)$, 第二个条件为 xs 大于 `stp`, 当这两个条件都满足时, 将执行循环体语句, 修改 s 点的坐标, 其修改语句为将 s 点坐标迭代为原坐标的 $\frac{1}{2}$, 同时更新 $f(s)$ 的值。当循环条件不满足时, 要么 $f(s)$ 大于 $f(x)$ 的假设条件成立, 要么 s 点已无限接近于零点, s 点与 x 点可近似看作一个点。

接下来的 `if` 语句判断了 `while` 循环停止的原因, 如果此时 s 点可看作零点, 则不变动初始值, 将参数 x 返回; 否则进入下一个 `else` 语句, 使用同样的方法寻找小于 $f(s)$ 的 $f(z)$, 此时函数停止循环的条件为循环体循环了 100 次, 如果循环 100 次后还未找到一个 z 点使得 $f(z)$ 小于 $f(s)$, 则令 $x+xs*\text{grad.f}x$ 为返回值。

当 s 点的位置不是 1 时, 显然, $f(1)$ 小于 $f(0)$, s 迭代到了 0.5 或更小的数上去, 此时令 z 为 s 的 2 倍值, $f(z)$ 小于 $f(x)$, 故它必然小于 $f(s)$, 但如果 $f(1)$ 大于 $f(0)$, 那我们对 $f(2)$ 及更远的点并不了解, 因此将 $gs < gz$ 作为循环条件是必要的。

当 x 、 s 、 z 点都确定后, 即可调用 12.2 节中出现的 `section` 函数使用黄金分割法求出局部最大值, 即 α 的值。由于并未绘出基础图形, 因此需将 12.2 节 `section` 函数中的 `points()` 函数去掉, 否则将报错。此时返回了 $x+\alpha*\text{grad.f}x$ 。

12.4.2 构建最快上升法函数并检验

函数 `fit.alpha` 实际上完成的是 x 的一次迭代, 即将 $x(n)$ 迭代到 $x(n+1)$ 上去。准备好 `fit.alpha` 函数后, 还需构建具体的最快上升法函数。

```
> fast <- function(f, grad.f, x, stp=1e-5){
+   x.old <- x; x <- fit.alpha(f, x, grad.fx=grad.f(x)) n <- 0;
+   while((f(x)-f(x.old)>stp) & (n<100)){
+     x.old <- x; x <- fit.alpha(f, x, grad.fx=grad.f(x)); n <- n+1
+   }
+   if(n>=100){print("can't find a right result")}
+   else{return(x)}
+ }
```

上述代码构造了函数 `fast`, 它接受 4 个传入参数, 其中 f 、 grad.f 、 x 分别代表原函数、梯度函数和初始值的坐标, 参数 `stp` 则给出了一个最大误差。

函数体的第 1 行代码将 x 赋给了 `x.old`, 并使用 `fit.alpha` 函数使 x 完成了第一次迭代, 此时 `fit.alpha` 将 x 更新为 x 、 $x+xs*\text{grad.f}x$ 或 $x+\alpha*\text{grad.f}x$ 中的某一个值, 总之, 将 x 更新为 x 上升速度最快的方向上最“高”的一个点。变量 n 记录了迭代的次数。

`while` 循环有两个循环条件, 只有在 $f(x)$ 与 $f(x.\text{old})$ 的差大于最大误差及迭代次数小于 100 次时, 循环体语句才会执行。在循环体语句中, `x.old` 更新为 x , x 则更新为函数 `fit.alpha` 计算出的新值, 循环体语句同时也增加了 n 的值。在循环结束后, 函数 `fast` 根据循环的次数返回了提示信息或 x 的值。

```
> fun <- function(x){-x[1]^2/2-x[2]^2/4}
> gf <- function(x){rbind(-x[1]-x[2]^2/4, -x[1]^2/2-x[2]/2)}
> fast(f=fun, grad.f=gf, x=c(1,0))
```

```
[,1]
[1,] 0.0002824211
[2,] -0.0007108237
```

上述代码中前两条命令给出了原函数 `fun` 和梯度函数 `gf` 的表达式，它们都接受一个向量 `x` 作为传入参数，其中，原函数的表达式为 $-\frac{x_1^2}{2} - \frac{x_2^2}{4}$ ，梯度函数的表达式为 $-x_1 - \frac{x_2^2}{4}$ 和 $-\frac{x_1^2}{2} - \frac{x_2^2}{2}$ ，我们所需要的梯度向量是一个列向量，`rbind()` 函数将这两个值按行连接为一个列向量。

最后一行调用了 `fast` 函数，并指定 `fast` 函数的参数 `f` 和 `grad.d` 分别为 `fun` 和 `gf`，`x` 的初始坐标指定为 $(1, 0)$ ，`fast` 函数给出的解为 $(0.000\ 282\ 421\ 1, -0.000\ 710\ 823\ 7)$ ，这是一个相当接近正确解 $(0, 0)$ 的解，但它的两个坐标与 $(0, 0)$ 的误差仍大于 $0.000\ 01$ ，即大于我们设置的最大误差。这意味着最快上升法在正确解附近求出了两个非常接近的解，因此得到了误差比我们想象的更大的返回结果。

对于同一个问题，最快上升法需要比牛顿迭代法更多的迭代次数才能求出结果，这是由于在每次迭代时最快上升法都追求当前迭代的方向是上升最快的方向，但该方向并不一定与最终结果的方向一致，因此最快上升法的迭代道路要比牛顿迭代法曲折得多，它的计算速度也比牛顿迭代法慢得多。

此外，本节给出的函数仅能解决求解最大值的问题，当求解最小值时，需要在函数前加一个负号。比如本节求解的是函数 $-\frac{x_1^2}{2} - \frac{x_2^2}{4}$ 的最大值，显然，也可看作求解的是 $\frac{x_1^2}{2} + \frac{x_2^2}{4}$ 的最小值。而牛顿迭代法则没有这样清楚地划分，有时牛顿迭代法会迭代出最大值，有时又会迭代出最小值，这取决于牛顿迭代法中初始坐标的选择。

12.5 R 中的最优化函数

关于函数的最优化，除本章介绍的三种较为简单的最优化方法外，R 还提供了内置的最优化函数，它们存储于 R 的基础包中，使用时只需直接调用即可。

关于一维情形下的最优化，R 提供了 `optimize()` 函数用于处理数据。

```
> fun <- function(x){-(print(x)-5)^2+25}
> optimize(fun,c(0,10),maximum=TRUE)
[1] 3.81966
[1] 6.18034
[1] 7.63932
[1] 5
[1] 4.999959
[1] 5.000041
[1] 5
$maximum
[1] 5
```

```
$objective
[1] 25
```

上述代码首先创建了一个函数 `fun`，它接受一个参数 x ，并以函数 $-(x-5)^2+25$ 作为被优化对象。在函数 `fun` 中，`print(x)` 表示每次调用函数 `fun`，并执行该部分代码时，都会在屏幕上输出当前的 x ，但这并不影响函数的计算。

第 2 行命令调用了 `optimize()` 函数，它接受三个参数，第一个参数 `fun` 指定被优化的对象；第二个参数则表示被优化的范围是 0~10，这里给出的是 x 的范围；`optimize()` 函数默认求解被优化函数的最小值，第三个参数则指定 `maximum` 为真，即求解被优化函数的最大值。

R 首先返回了 `print(x)` 的值，`optimize()` 函数从 3.819 66 开始迭代，经过 7 次迭代后达到最大点 5，`maximum` 返回了最终的结果，`objective` 则返回了最大解对应的函数值。显然，`optimize()` 函数求出了正确的结果。

```
> fun <- function(x,a){-(print(x)-5)^2+a}
> optimize(fun,c(0,10),tol=1e-5,maximum=TRUE,a=25)
[1] 3.81966
[1] 6.18034
[1] 7.63932
[1] 5
[1] 4.999997
[1] 5.000003
[1] 5
$maximum
[1] 5

$objective
[1] 25
```

`optimize()` 函数仅能处理一维的情况，不过它也能接受多个函数参数的传入。上述代码将函数 `fun` 修改为 $-(x-5)^2+a$ ，此时 `fun` 同时接受参数 x 和 a 。与上一个小例子相比，此时 `optimize()` 函数中增加了两个参数，其中参数 a 指定函数 `fun` 中的 a 取值为 25，由于 a 的值被固定，因此此时 `optimize()` 函数处理的还是一维的情形。将这种形式与循环语句结合起来，`optimize()` 函数便可处理多维的情形。

除此之外，上述代码还添加了参数 `tol`，它用于指定实际解与近似解的误差。观察 R 返回的结果，`optimize()` 函数同样选择 3.819 66 作为初始坐标，经过 7 次迭代后达到最优解。`optimize()` 函数的初始坐标并不是随意选择的，`optimize()` 函数结合使用了黄金分割法和抛物线插补法两种算法，与前面的黄金分割法相比，`optimize()` 函数只需设定 x 点、 z 点的坐标即可，而不需要知道 s 点的坐标。

```
> fun <- function(x){x[1]^2/2+x[2]^2/4}
> optim(c(1,0),fun)
$par
```

```
[1] -8.458290e-06 -4.430304e-05

$value
[1] 5.264612e-10

$counts
function gradient
      87      NA

$convergence
[1] 0

$message
NULL
```

关于多维情形，R 提供的函数为 `optim()` 函数。上述代码首先创建了被优化的函数 `fun`，函数 `fun` 的表达式为 $\frac{x_1^2}{2} + \frac{x_2^2}{4}$ ，它接受一个向量形式的参数 x 。第二行命令调用了 `optim()` 函数，其中第一个参数指定 x 的初始坐标为 $(1, 0)$ ，第二个参数指定被优化的函数为 `fun`。

R 返回了最优化的结果，`par` 给出了最优解的坐标，这是一个无限近似于 $(0, 0)$ 的坐标，`value` 则给出了最优解对应的函数值。`optim()` 函数自动求解了函数的最小值，如果在被优化函数前加上负号，那么 `optim()` 函数将给出一个尽可能大的坐标，以求出尽可能小的函数值。

`optim()` 函数还提供了 `method` 参数用以选择优化函数的算法，上述代码并未设定 `method`，故 `optim()` 函数使用了默认的内尔德-米德算法作为最优化算法，其他可用的算法还有准牛顿算法、共轭梯度法和模拟退火等方法，它们都是针对多维情形设计的最优化算法。此外，`optim()` 函数也提供了 `Brent` 选项，当 `method` 设定为 `Brent` 时，`optim()` 函数的作用与 `optimize()` 函数相同。

第 13 章 使用 R 绘制地理信息图形

本章的主题是地理信息图形的绘制，也就是地图的绘制。地图是一种非常特别的图形，它在表现空间信息时非常重要，地图也是数据可视化的一项重要内容。本章将介绍如何使用 R 绘制漂亮的图形，以及解读地图时的一些要点。通过阅读本小节，读者也将踏入数据可视化的大门。

13.1 绘制世界、国家、省市地图

本节将介绍两种绘制基础地图的方法，还将讨论一些有关地图绘制的专业地理知识。阅读本节的内容是学习后续章节的必要准备工作。

13.1.1 使用 `map()` 函数绘制地图

有关地图的数据可视化如今已达到非常精美的地步，有些 R 做出的地图简直美得让你不相信那是地图。但是我们应当知道，再精美的地图也是由简单的地图一点一点叠加、变形得来的，而最简单的地图就是那种用线条勾出边界线的地图。

在正式绘图之前，有必要先介绍在绘制地图时常用的数据类型。地图数据主要分为点、线、面三种，一个点可由一组坐标给出，一条线是无数个点的集合，一个面则是无数个线的组合，实际情况中我们无法取出无数组坐标，但可以想象，要绘制一幅包含许多线条的地图需要用到许多组坐标，这个坐标数要远多于绘制一幅二次函数或者其他图形。

```
> library(maps)
> map("world", fill=TRUE, col=rainbow(50))
> map("state", fill=TRUE, col=rainbow(50))
```

程序包 `maps` 内置了世界地图和美国地图的坐标数据，上述代码加载了这个包，并使用两条 `map()` 函数分别绘出了世界地图和美国地图，在两条命令中，参数 `fill` 被指定为真，即填充每一“块”地图，参数 `col` 则指定了使用 `rainbow(50)` 填充地图，即 50 种彩虹色。

上述代码在绘制美国地图时调用了 `state` 数据集，如果调用 `usa` 数据集，则只能画出一整个美国轮廓，而不会划分出每个具体的州。

```
> library(mapdata)
> map("china", ylim = c(18, 54))
```


让人遗憾的是程序包 `maps` 中并未内置除美国之外的其他国家坐标数据集，为了绘制中国地图，上述代码加载了程序包 `mapdata`，第 2 行代码调用了 `map()` 函数，并绘出了一张中国地图，`map()` 函数中的参数 `ylim` 缩小了默认的纵坐标轴范围，实际上，在绘制世界地图和美国地图时，`map()` 函数就已经倾向于将图形压得扁扁的，这里我们不得不调整纵坐标轴范围，以保证地图不会太走样。

`mapdata` 中的数据年份较为久远，并不十分精准。除中国地图外，`mapdata` 也提供日本数据、新西兰数据、世界河流数据等十多份其他各类数据以供绘制地图。

13.1.2 另一种绘制地图的方法

关于绘制地图，R 能够提供的内置数据毕竟非常有限，另一种更常用的方法就是自行向 R 中上传坐标数据，并根据坐标数据绘制图形。仍以绘制中国地图为例，这种数据一般放在国家基础地理信息数据包中，如今虽然国家基础地理信息网不再提供下载，但网上仍能很方便地下载到。

地理信息数据有其特有的格式，`dbf`、`shp`、`shx` 是最为常用的三种数据格式，程序包 `maptools` 提供了一些有关读取这三种数据的函数。

```
> library(maptools)
> map.shp <- readShapePoly("boul_4p.shp")
> plot(map.shp)
```

上述代码首先加载了 `maptools` 程序包，然后调用 `readShapePoly()` 函数读取了 `boul_4p.shp` 数据集，这是一份“面”类型的图形数据，即使用多边形勾勒出边界，类似地还有读取点数据的 `readShapePoints()` 函数和读取线数据的 `readShapeLines()` 函数等。在读取文件前，需要提前将数据文件放在 R 的工作目录下，还需将扩展名为 `dbf` 和扩展名为 `shx` 的辅助文件放入工作目录下，否则 R 将无法读取数据文件。

`plot()` 函数使用从 `shp` 文件中读取的数据绘出了地图，它同样是一张有些扁平的图形，这是由于 `plot()` 函数将地图坐标默认绘制在普通的坐标系中，但地球是一个球形，因此将地图坐标画在普通坐标系中时地图将被压扁。

```
> library(ggplot2)
> map.for <- fortify(map.shp)
> head(map.for)
      long      lat order  hole piece group id
1 123.0003 39.27521     1 FALSE     1    0.1  0
2 123.0057 39.27477     2 FALSE     1    0.1  0
3 123.0145 39.27732     3 FALSE     1    0.1  0
4 123.0231 39.27896     4 FALSE     1    0.1  0
5 123.0337 39.28060     5 FALSE     1    0.1  0
6 123.0425 39.28086     6 FALSE     1    0.1  0
```

为了修正地图，首先需要将地图坐标数据转换为普通的数据框，以便于对坐标数据进行其他处理。上述代码加载了程序包 `ggplot2`，并调用了 `fortify()` 函数将 `map.shp` 转换

为矩阵类型的 `map.for`，`head()` 函数查看了 `map.for` 的前 6 行，它共有 7 列，前两列是每个点的坐标信息，这些坐标差异非常微小，显然，这些点挨得非常近，第 3 列 `order` 则给出了连接坐标点的顺序。

```
> library(mapproj)
> map.china <- ggplot(data=map.for)+
+ geom_polygon(aes(x=long, y=lat, group=id), col="black",fill=NA)+
+ coord_map()
> map.china
```

上述代码首先加载了 `mapproj` 程序包，第二个语句使用一个复合语句给出了绘图信息，其中 `ggplot()` 函数指明被绘制的对象为数据框 `map.for`；`geom_polygon()` 函数则表明要绘制一个多边形，其中参数 `aes` 给出了多边形的横、纵坐标，`group` 则表明按照 `id` 分组绘制多边形，参数 `col` 和 `fill` 则表明使用黑色线条绘制地图，且不要填充它；`coord_map()` 函数来自 `mapproj` 程序包，它表示用专业的地理坐标系绘制地图。这三个图层组合得到的结果存储在 `map.china` 中，输入命令 `map.china`，便得到了带有地理坐标的地图。

此外，`ggplot` 函数自动为地图添加了灰色的背景，一些横线和竖线也交叉在图形中，这些线条对应真正的经纬度，给出了中国在地球上的真实位置。`coord_map()` 函数默认的绘图类型为 `mercator`，在 `coord_map()` 函数中将参数 `projection` 指定为其他值，也可得到其他类型的专业地理图形。

13.1.3 分省市绘制地图

现在我们已经学习了如何使用 R 自带的坐标数据或使用 R 读取本地坐标数据并绘制地图，有时我们仅对地图中的某一部分感兴趣，比如有时我们只想绘制某一个或某几个省的地图，`map()` 函数准备了 `region` 参数以供绘制地图的某一部分。

```
> library(maps)
> library(mapdata)
> library(maptools)
> library(ggplot2)
> map('state', region=c('new york', 'new jersey', 'penn'),fill=TRUE,
col=rainbow(3))
```

上述代码首先加载了 4 个程序包，然后调用了 `map()` 函数以绘制地图，`map()` 函数中参数 `state` 表明图形数据来源为 `state`，参数 `region` 则表明仅绘制 `state` 的纽约州、新泽西州和宾夕法尼亚州三个地方。参数 `fill` 和 `col` 使用了三种彩虹色填充了这三个州。类似地，也可以绘出中国的省市地图。

使用下载好的本地坐标数据绘制省市地图时，要比使用 R 内置数据复杂一些。在绘制省市地图之前，首先需要将数据从数据集中抽取出来。

```
> map.shp <- readShapePoly("BOUNT_poly.shp")
> tmp <- map.shp$NAME99
> grep("成都", tmp, value = TRUE)
```

```
[1] "成都市市辖区" "成都市市辖区" "成都市市辖区"
> grep("成都", tmp)
[1] 1657 1678 1697
```

上述代码使用 `readShapePoly()` 函数读取了有关省市坐标的面数据，并将它们存储在 `map.shp` 中，`map.shp` 中包含一个变量 `NAME99`，它给出了中国所有省市的名称，第 2 行命令将它赋给了 `tmp`。

第 3 行命令使用 `grep()` 函数抽取了 `tmp` 中所有包含“成都”字样的元素，参数 `value` 指定为真，即返回每一个元素的具体信息，R 返回了三个“成都市市辖区”，表明成都市共有三个市辖区。第 4 行命令去掉了第 3 行命令中的 `value` 参数，此时返回的是三个“成都市市辖区”分别对应的行数。

```
> map.shp$ADCODE99[grep("成都", tmp)]
[1] 510101 510101 510101
2368 Levels: 0 110100 110112 110113 110221 110224 110226 110227 110228
110229 120100 120221 ... 820000
> Chengdu <- map.shp[substr(as.character(map.shp$ADCODE99), 1, 4) ==
"5101",]
```

知道了三个“成都市市辖区”所处的位置后还不够，还需查询出它们的 `ADCODE99` 编码。上述代码查看了 `map.shp` 中变量 `ADCODE99` 的第 1 657、第 1 678、第 1 697 个元素，即三个“成都市市辖区”的 `ADCODE99` 编码。使用 `readShapePoly()` 读取 `shp` 文件将得到一个 `sp` 类型的变量，它和矩阵类似，`[]` 符号和 `$` 符号能对其实现与矩阵相同的功能。

根据 R 返回的结果，三个成都市市辖区的 `ADCODE99` 编码都是 510101，它由省、市、县各两位编码组成，显然，51、01、01 分别表示四川省、成都市和市辖区。成都市除三个市辖区外，还包含其他一些地级市和一些区县，它们的最末两位编码则不为 01。

最后一行命令在 `map.shp` 中抽取了所有 `ADCODE99` 编码中包含 5101 的数据，并赋给了变量 `Chengdu`，在给出抽取条件时，`as.character()` 函数首先将 `ADCODE99` 变量从因子型转换为字符型，`substr()` 函数则表示抽取条件为 `ADCODE99` 变量的前 4 个字符为 5101 的全部数据。

此时成都市的全部数据已经抽取完毕，使用 `plot()` 函数为 `Chengdu` 绘制图形，将得到一张成都市的地图，使用 `plot()` 函数绘出的地图将呈现一种被压扁的形状，为了绘制更标准的地图，还需进一步处理数据。

```
> mpsh <- fortify(Chengdu)
> qplot(long,lat,data=mpsh)+geom_map(aes(map_id=id,),color="black",fill=NA,
map=mpsh)+coord_map()
```

上述代码使用 `fortify()` 函数将 `Chengdu` 转变为矩阵类型的 `mpsh` 变量。最后一行代码则使用了程序包 `ggplot2` 提供的绘图函数，`qplot()` 函数给出了图形的横、纵坐标轴及数据来源；`geom_map()` 函数则表明绘制一张地图，其中参数 `map_id` 是地图中多边形分组的依据，参数 `map` 则是数据来源，参数 `fill` 和 `col` 共同绘出了一张不进行填充、用黑色线条勾边的地图；`coord_map()` 函数表示在地理坐标系上绘制地图。

13.2 向地图中添加颜色

本节将讨论如何向地图中添加颜色，这种简单的功能实现起来并不简单。通过学习读本节内容，读者也将对地图数据结构有更深一步的认识。

13.2.1 向地图中添加颜色前的准备工作

彩色的地图不仅有美化图形的作用，很多时候色彩也能传递非常重要的信息。

```
> library(maps)
> library(mapdata)
> library(maptools)
> library(ggplot2)
```

向地图中添加色彩需要一些烦琐的准备工作，上述代码加载了 `maps`、`mapdata`、`maptools` 和 `ggplot2` 四个与绘地图有关的程序包，这是本节用到的程序包，此处将其全部加载完毕。

```
> map.shp <- readShapePoly("bou2_4p.shp")
> map.for <- fortify(map.shp)
> head(map.for)
      long      lat order  hole piece group id
1 121.4884 53.33265     1 FALSE     1   0.1  0
2 121.4995 53.33601     2 FALSE     1   0.1  0
3 121.5184 53.33919     3 FALSE     1   0.1  0
4 121.5391 53.34172     4 FALSE     1   0.1  0
5 121.5738 53.34818     5 FALSE     1   0.1  0
6 121.5840 53.34964     6 FALSE     1   0.1  0
> nrow(map.for)
[1] 91040
```

仍以中国省份地图为例，上述代码中第 1 行代码使用 `readShapePoly()` 函数加载了 `bou2_4p.shp` 数据文件，第 2 行代码则使用 `fortify()` 函数将 `sp` 类型的变量 `map.shp` 转换为矩阵类型的变量 `map.for`。`head()` 函数查看了 `map.for` 中的信息，`nrow()` 函数则查看了它的行数。

观察 R 返回的信息，`map.for` 是一个由 91 040 行数据构成的矩阵，每一行数据就是一个坐标点，每一个省份都需要上千个坐标点勾出形状。如果仅需要绘出一张由线条构成的地图，`map.for` 中的 `long` 向量和 `lat` 向量便可完成这项任务，但如果想为每个省份添加不同的颜色，就需要一个能够区分每个坐标点对应的省份的变量。

```
> summary(unique(map.for$id))
      Length      Class      Mode 
      925 character character
```

矩阵 `map.for` 中的变量 `id` 能起到这个作用，它给出了每一个坐标所属省份的序号，

这些坐标实际上都位于两个省份的交界处，但 `map.for` 为每个坐标划分了一个省份归属，这给我们为每个省份添加颜色提供了方便。

`unique()` 函数能够提取出向量中所有不同的元素，上述代码对 `map.for` 中变量 `id` 应用了 `unique()` 函数，`summary()` 函数则查看了它的摘要。R 返回的结果显示变量 `id` 中一共有 925 个不同的元素，这个数字仍比中国的省份数目大许多，这是由于 `id` 变量给出的实际上是图形数据中多边形的序号，比如，海南省包含许多分散的岛屿，那么海南省将由好几个多边形组成，也就有好几个 `id` 对应着海南省。中国沿海大大小小的诸多岛屿最终组成了 925 个多边形。

```
> rnf <- runif(925)
> vals <- data.frame(id =unique(map.for$id),val=rnf)
> head(vals)
  id      val
1  0 0.5786881
2  1 0.7716431
3  2 0.6766779
4  3 0.8975300
5  4 0.9988538
6  5 0.1647081
```

这 925 个多边形需要使用不同的颜色，因此还需要一个长度同样为 925 的向量用于给出绘图颜色的标准。上述代码首先使用 `runif()` 创建了 925 个服从均匀分布的随机数，并将这 925 个随机数赋给了变量 `rnf`，第 2 行代码则将 `unique(map.for$id)` 和 `rnf` 组合为数据框 `vals`，并分别为其命名为 `id` 和 `val`。

`head()` 函数查看了 `vals` 的前 6 行，`id` 是一个从 0 开始、逐一递增的向量，它代表了 925 个多边形，`val` 则是一个随机分散在 0 和 1 之间的绘图指标，我们将根据 `val` 的大小决定对应的多边形应当绘成什么颜色。

13.2.2 在地图上添加颜色

在地图上添加颜色前，最后一步准备工作是修改 `map.for` 的变量名称，这一步是为调用 `geom_map()` 函数绘图做准备。

```
> colnames(map.for) <- c("x","y","order","hole","piece","group","id")
```

上述代码把 `map.for` 中前两列变量的名字修改为 `x` 和 `y`，即把 `map.for` 中绘图时使用的横坐标命名为 `x`，纵坐标命名为 `y`，这与使用 `geom_map()` 函数绘图时默认的坐标变量名称保持了一致。

```
> ggplot(vals) +
+ geom_map(aes(map_id=id, fill=val), color="white", map=map.for) +
+ scale_fill_gradient(low="red2",high="yellowgreen") +
+ expand_limits(map.for) + coord_map()
```

上述代码使用 5 个函数组合绘出了地图，函数 `ggplot()` 表明 `vals` 为作图对象，我们需要使用 `vals` 中的 `id` 信息及其对应的 `val` 值确定每个省份的颜色，故 `vals` 是作图的主体。

`geom_map()` 函数中参数 `aes` 指定按照 `id` 区分省份，并按照 `val` 的值填充颜色，参数 `color` 指定每个省份的边界线为白色，参数 `map` 则指定绘图数据来自 `map.for`，这意味着将从 `map.for` 中抽取绘图坐标数据，由于之前已经修改了 `map.for` 中向量的名称，故 `geom_map()` 函数会直接将 `map.for` 中名称为 `x` 的向量看作横坐标，名称为 `y` 的向量看作列坐标。

`val` 值仅给出了一个绘图颜色的深浅标准，`scale_fill_gradient()` 函数指定了具体的绘制颜色，参数 `low` 表示 `val` 值越小，对应省份的颜色越接近 “red2”；参数 `high` 表示 `val` 值越大，对应省份的颜色越接近 “yellowgreen”。

`expand_limits()` 函数较为容易地将 `map.for` 绘在了 `vals` 上，这个函数起到连接 `vals` 和 `map.for` 的作用，`coord_map()` 函数则指定在地理坐标系上绘制地图。

利用下面的代码，将每一个省份都标出了不同的颜色，图形右侧的图例则给出了判断标准，`val` 的值越小，省份的颜色就越红，`val` 的值越大，省份的颜色就越偏向黄绿色。图中省份之间的边界线为白色，这弱化了边界线的存在，使得省份本身的颜色更加显眼。

```
> provcol <- rgb(red=1-val/max(val),green=0,blue=0)
> ggplot(vals) +
+ geom_map(aes(map_id = id,fill=provcol), color = "white", map = map.for) +
+ expand_limits(map.for) + coord_map()
```

另一种为省份添加颜色的方法是使用 `rgb` 系统，`rgb` 是 `red`、`green`、`blue` 的缩写，它表示用三原色混合调配颜色，上述代码中第 1 行命令根据 `val` 的值生成了一系列 `rgb` 值，`val` 值越大，`val/max(val)` 就越接近 1，`red` 也就越接近 0。

设置变量 `provcol` 后，接下来的命令调用了 `ggplot()` 函数、`geom_map()` 函数、`expand_limits()` 函数和 `coord_map()` 函数绘出了不同颜色的地图。`geom_map()` 函数中的 `fill` 参数指定为 `provcol`，由于它本来就代表一系列颜色，故无须再调用 `scale_fill_gradient()` 函数为省份添加颜色。

13.3 向地图中添加标签和线条

本节将介绍如何向地图中添加标签和线条。本节的主题与 13.2 节互相呼应，都是绘制地图的基本方法，本节还将讨论更多绘制地图的细节。

13.3.1 向地图中添加标签前的准备工作

向地图中添加标签仍用到了 `maps`、`mapdata`、`maptools` 和 `ggplot2` 四个程序包，在上文中已经加载过这四个包，此处无须再次加载。向地图中添加标签是另一种丰富地图信息的常用方法，最常见的标签就是各省的省会名称。

```
> map.shp <- readShapePoly("bou2_4p.shp")
> res.shp <- readShapePoints("res1_4m.shp")
> map.for <- fortify(map.shp)
```

与向地图中添加颜色类似，向地图中添加标签也需要用到两个文件。国家基础地理信息网同样提供了有关省会坐标的文件。上述代码分别读取了 `bou2_4p.shp` 文件和 `res1_4m.shp` 文件。`bou2_4p.shp` 文件已经用到过很多次，它是有关中国省份的“面”类型文件，`readShapePoly()` 函数能够读取其中的信息。`res1_4m.shp` 文件则存储了省会坐标，这是一个“点”类型的文件，因此要用 `readShapePoints()` 文件读取它。

上述代码还使用 `fortify()` 函数将 `map.shp` 转换为矩阵 `map.for`，但 `fortify()` 函数仅能处理“面”类型的文件，而不能处理“点”类型的文件，还需对 `res.shp` 进行其他处理。

```
> class(res.shp)
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
> head(res.shp)
```

	coordinates	AREA	PERIMETER	RES1_4M_	RES1_4M_ID	GBCODE	NAME
0	(116.3809, 39.92361)	0	0	1	61	31010	北京
1	(117.2035, 39.13112)	0	0	2	70	31020	天津
2	(114.4898, 38.04513)	0	0	3	121	31020	石家庄
3	(112.5694, 37.87111)	0	0	4	220	31020	太原
4	(111.6633, 40.82094)	0	0	5	322	31020	呼和浩特
5	(123.4117, 41.79662)	0	0	6	412	31020	沈阳

```
ADCODE93 ACD93 ACD99 ADCLASS PINYIN
0 110100 110100 1 Beijing
1 120100 120100 2 Tianjin
2 130101 130101 2 Shijiazhuang
3 140101 140101 2 Taiyuan
4 150101 150101 2 Huhehaote
5 210101 210101 2 Shenyang
```

上述代码首先使用 `class()` 函数查看了 `res.shp` 的类型，它有两个属性，分别是 `SpatialPointsDataFrame` 和 `sp`，第一个属性表明 `res.shp` 是一种类似数据框的结构，使用 `head()` 函数查看它的前 6 个元素，显然，它确实是一张数据框类型的表格，但如果用 `head()` 函数查看 `map.shp`，R 就会刷屏。

观察 R 返回的结果，其中，向量 `coordinates` 给出了我们所需要的坐标信息，向量 `NAME` 则给出了每一个坐标对应的中文名称，向量 `PINYIN` 给出了中文名称的拼音，其他的向量则是一些编码信息。`res.shp` 并不能直接添加到绘图函数中，`ggplot2` 包不能直接处理 `SpatialPointsDataFrame` 类型的数据，此外，每一个省会的横、纵坐标存储在一个向量中，我们希望把它们分开以便于分别调用它们。

```
> res.shp <- as.data.frame(res.shp)
> head(res.shp)
```

	AREA	PERIMETER	RES1_4M_	RES1_4M_ID	GBCODE	NAME	ADCODE93	ADCODE99
0	0	0	1	61	31010	北京	110100	110100
1	0	0	2	70	31020	天津	120100	120100
2	0	0	3	121	31020	石家庄	130101	130101
3	0	0	4	220	31020	太原	140101	140101
4	0	0	5	322	31020	呼和浩特	150101	150101
5	0	0	6	412	31020	沈阳	210101	210101
	ADCLASS	PINYIN	coords.x1	coords.x2				
0	1	Beijing	116.3809	39.92361				
1	2	Tianjin	117.2035	39.13112				
2	2	Shijiazhuang	114.4898	38.04513				
3	2	Taiyuan	112.5694	37.87111				
4	2	Huhehaote	111.6633	40.82094				
5	2	Shenyang	123.4117	41.79662				

由于 `res.shp` 的结构与数据框非常类似，故 `as.data.frame()` 函数能够对其起作用，上述代码中第 1 行命令使用 `as.data.frame()` 函数将 `res.shp` 转换为数据框类型的变量，再次使用 `head()` 函数查看 `res.shp` 中的元素，此时 `res.shp` 已转换为一个数据框，它所存储的向量并未发生大的改动，只是 `coordinates` 向量被拆为 `coords.x1` 和 `coords.x2` 两个向量，并放在了数据框的最末两位，显然，`coords.x1` 是每个省会的横坐标，`coords.x2` 是每个省会的纵坐标。

13.3.2 在地图上添加标签

准备好地图坐标数据和省会坐标数据后，`ggplot2` 包提供的图层语言能够很方便地在之前绘好的中国地图上添加标签信息。

```
> ggplot(data=map.for)+
+ geom_polygon(aes(x=long, y=lat, group=id), col="black",fill=NA)+
+ geom_text(aes(x=coords.x1,y=coords.x2,label=NAME),data = res.
shp,size=3)+
+ coord_map()
```

上述代码中 `ggplot()` 函数和 `geom_polygon()` 函数绘出了一张中国地图，`coord_map()` 函数表明图形绘制在地理坐标系中，这三个函数的用法在上文中已经介绍过。

`geom_text()` 函数则是一个新函数，它负责在图形中添加文本信息，其中参数 `aes` 指定被添加文本以 `coords.x1` 为横坐标，以 `coords.x2` 为纵坐标，以 `NAME` 为文本内容，参数 `data` 指定 `res.shp` 为被添加文本的数据来源，参数 `size` 则指定文本的大小，`size` 越大，文本就越大。

在 `geom_text()` 函数中，使用 `size` 参数能够调整文本的大小，但实际上当所有的文本都独立开来时，它们会太小而不能被清楚地辨认出来。

```
> ggplot(data=map.for)+
+ geom_polygon(aes(x=long, y=lat, group=id), col="black",fill=NA)+
```



```
+ geom_text(aes(x=coords.x1,y=coords.x2,label="+"),data = res.shp)+
+ coord_map()
```

上述代码修改了前面的代码，它将 `geom_text()` 函数中的 `label` 参数由 `NAME` 改为了符号“+”，所有的省会城市坐标都用一个“+”标出。

此时不需要再担心省会坐标被彼此遮住，而且符号“+”也能更精确地标出省会的位置。但图中仍存在一些“+”被省界线挡住，不妨调整省界线的颜色以进一步改善这张地图。

```
> ggplot(data=map.for)+
+ geom_polygon(aes(x=long, y=lat, group=id), col="grey",fill=NA)+
+ geom_text(aes(x=coords.x1,y=coords.x2,label="+"),data = res.shp,)+
+ theme_bw()+coord_map()
```

上述代码进一步调整了 `geom_polygon()` 函数，参数 `col` 被改为“grey”，即绘制灰色的边界线，这将能够在地图上凸显出黑色的“+”符号。考虑到 `ggplot2` 默认的绘图背景为灰色，为了不让背景淹没灰色的边界线，上述代码在最后添加了 `geom_polygon()` 函数，它将背景颜色设置为白色。如果要在每个“+”符号旁添加中文名称，使地图达到专业地图的标准，那么一种可行的办法是再次添加一个 `geom_text()` 函数，用于加入中文名称。新的 `geom_text()` 函数显然不能直接使用 `res.shp` 提供的坐标，在该坐标的基础上进行微调，从而使生成的省会名称恰好既不相互遮挡，也不遮挡住“+”符号。

13.3.3 在地图上添加线条

在地图上添加线条与在地图上添加点的工作原理类似，在地图上添加许多个点并将它们缀连起来，便得到了一根线条。

```
> map.shp <- readShapePoly("boul_4p.shp")
> hyd.shp <- readShapeLines("hyd1_4l.shp")
```

上述代码读取了两份数据，其中，`boul_4p.shp` 是中国的国界线，它不包含省界线；`hyd1_4l.shp` 是中国的一级河流的坐标数据，它包含了中国的全部一级河流。我们准备使用这两份数据绘出一张中国的一级河流分布图。

```
> map.for <- fortify(map.shp)
> hyd.for <- fortify(hyd.shp)
> head(map.for,n=2)
      long      lat order piece group id
1 123.0003 39.27521     1     1   0.1  0
2 123.0057 39.27477     2     1   0.1  0
> head(hyd.for,n=2)
      long      lat order piece group id
1 117.7539 49.16444     1     1   0.1  0
2 117.7578 49.16217     2     1   0.1  0
```

上述代码使用 `fortify()` 函数将 `map.shp` 和 `hyd.shp` 分别转换为矩阵 `map.for` 和 `hyd.for`。`fortify()` 函数既可以转换“面”类型的坐标数据,也可以转换“线”类型的坐标数据,因此, `map.shp` 和 `hyd.shp` 都可以使用 `fortify()` 函数转换。

上述代码的第 3、4 行命令使用 `head()` 函数分别查看了 `map.for` 和 `hyd.for` 的前两行,观察 R 的返回结果, `map.for` 和 `hyd.for` 有非常相似的数据结构, `long` 和 `lat` 分别为每一个坐标数据的横轴坐标和纵轴坐标。

向量 `id` 则区分了每个坐标所属的组。`map.for` 代表国界线坐标,因此它的 `id` 应当大部分都是 0,代表中国的国界线,另外的一些 `id` 则对应中国周边海岛的坐标。`hyd.for` 代表一级河流坐标,它的 `id` 则用于区分不同河流的坐标数据。

```
> ggplot(map.for)+
+ geom_polygon(aes(x=long, y=lat, group=id),fill=NA,col="grey")+
+ geom_line(aes(x=long,y=lat,group=id,col=id),data=hyd.for)+
+ coord_map()+theme_bw()
```

上述代码调用了 5 个函数, `ggplot()` 函数表明绘图主体为 `map.for`。`geom_polygon()` 函数绘出了一个多边形图层,其中,参数 `aes` 指定使用 `long` 作为横坐标,使用 `lat` 作为纵坐标,并使用 `id` 为坐标数据分组,参数 `fill` 和 `col` 共同作用,指定图形边界为灰色,且不使用颜色填充,函数并未特别指明 `data` 参数,因此 R 将会从 `map.for` 中抽取 `long`、`lat` 和 `id` 向量。`geom_line()` 函数绘出了第二个图层,参数 `aes` 同样指定使用 `long` 作为横坐标,使用 `lat` 作为纵坐标,并使用 `id` 为坐标数据分组。此外,参数 `col` 将为 `id` 不同的坐标绘出不同的颜色,即不同的一级河流将有不同的颜色。参数 `data` 表明使用 `hyd.for` 中的数据绘制这个图层。`coord_map()` 函数和 `theme_bw()` 函数分别表明使用地理坐标系绘制图形,以及将背景颜色设置为空白,这可以绘出合理的地图,并使淡灰色的国界线显眼一些。

如果希望体现河流流经的省份区域,只需将 `bou1_4p.shp` 改为 `bou2_4p.shp` 文件即可。按照类似的思路,我们也能绘出中国的山脉分布图、公路分布图、铁路分布图等图形。

13.4 使用其他格式的文件优化地图

在向地图中添加点和线条时,实际上就是向地图中添加坐标点。在前几节中,我们使用的文件格式均为标准的地理图形文件 `shp` 格式,但能用于绘制地图的文件并不仅限于 `shp` 格式,所有包含坐标信息的文件都能用于绘制地图。

```
> library(maps)
> library(mapdata)
> jd <- c(116.4666667,121.4833333,117.1833333,106.5333333,126.6833333,125.
3166667,123.4,
+ 111.8,114.4666667,112.5666667,117,113.7,108.9,103.8166667,106.2666667,10
1.75,87.6,117.3,
+ 118.8333333,120.15,113,115.8666667,114.35,104.0833333,106.7,119.3,121.51
66667,113.25,
```

```

+ 110.3333333,108.3333333,102.6833333,91.16666667,114.1666667,113.5)
> wd <- c(39.9,31.23333333,39.15,29.53333333,45.75,43.86666667,41.83333333,
40.81666667,
+ 38.03333333,37.86666667,36.63333333,34.8,34.26666667,36.05,38.33333333,3
6.63333333,43.8,
+ 31.85,32.03333333,30.23333333,28.18333333,28.68333333,30.61666667,30.65,
26.58333333,
+ 26.08333333,25.05,23.13333333,20.03333333,22.8,25,29.66666667,22.3,22.2)
> name <- c("北 京","上 海","天 津","重 庆","哈尔滨","长 春","沈 阳","呼和浩
特",
,"石 家 庄",
+ "太 原","济 南","郑 州","西 安","兰 州","银 川","西 宁","乌鲁木齐","合 肥","南 京",
+ "杭 州","长 沙","南 昌","武 汉","成 都","贵 阳","福 州","台 北","广 州","海 口",
+ "南 宁","昆 明","拉 萨","香 港","澳 门")

```

直接利用坐标向地图中添加点是可行的，上述代码加载了 `maps` 程序包和 `mapdata` 程序包，并创建了三个向量 `jd`、`wd` 和 `name`，这三个向量分别是 34 个城市的经度、纬度和名称，这些信息来自中国基础地理信息网，三个变量具有相同的程度，彼此成一一对应的关系。

```

> city <- data.frame(jd,wd,name)
> head(city)
      jd      wd  name
1 116.4667 39.90000 北 京
2 121.4833 31.23333 上 海
3 117.1833 39.15000 天 津
4 106.5333 29.53333 重 庆
5 126.6833 45.75000 哈尔滨
6 125.3167 43.86667 长 春

```

上述代码使用 `data.frame()` 函数将 `jd`、`wd`、`name` 组合为一个数据框 `city`，`head()` 函数查看了 `city` 的前 6 个元素，显然，此时 `city` 中存储了 34 个点的坐标及每个点的名字，根据坐标信息，我们很容易能够将这 34 个点添加到一张地图中。

```

> map("china", col = "darkgray", ylim = c(18, 54), panel.first = grid())
> points(city$jd, city$wd, pch = 19)
> text(city$jd, city$wd, city$name, cex = 0.65,
+ pos = c(2, 4, 4, 4, 3, 4, 2, 3, 4, 2, 4, 2, 2, 4, 3, 2, 1, 3, 1, 1, 2,
3, 2, 2, 1, 2, 4, 3, 1, 2, 2, 4, 4, 2))
> axis(1, lwd = 0); axis(2, lwd = 0); axis(3, lwd = 0); axis(4, lwd = 0)

```

上述代码首先使用 `map()` 函数绘出了一张中国地图，它使用的国界线、省界线坐标由 `mapdata` 包提供，参数 `col` 指定绘制的线条颜色为黑灰色，参数 `ylim` 则限制了 `y` 轴的长度，最后一个参数 `panel.first=grid()` 给地图添加了一个带有灰色虚线的背景，这有助于确定每个点的具体坐标。

构建好背景后，`points()` 函数向图中添加了 34 个点，这些点以数据框 `city` 中的 `jd` 为

横坐标，以数据框 `city` 中的 `wd` 为纵坐标，参数 `pch` 指定为 19，即绘制实心小圆圈。这一行命令在图中添加了 34 个实心圆圈，它们标出了 34 个省会城市的具体位置。

`text()` 函数为每个点添加了名称，前三个参数分别表示使用数据框 `city` 中的 `jd`、`wd` 为横、纵坐标，并将 `name` 作为文本内容，参数 `cex` 表明文本的大小为默认大小的 0.65 倍。在绘制时可以发现直接在每个省会城市的坐标处添加城市名称会导致彼此遮挡住，为了避免这种情况，`text()` 函数中使用参数 `pos` 挪动了每个名称的位置，`pos` 的值为 1、2、3、4 时分别表示对应名称向上、左、下、右挪动一个单位，挪动的单位长度随文本的大小而改动。

最后一行命令使用 4 个 `axis()` 函数在图形的上、下、左、右分别添加了一条坐标轴。在运行结果中，每一个城市的名称都相互错开，而坐标轴则起到重要的辅助作用，地图达到了相对完美的程度。

```
> earthquake <- read.csv("earthquake.csv")
> head(earthquake)
```

	date	time	wd	jd	deep	level	valve	type	address
1	2015-10-19	13:42:14.8	24.97	121.79	7	Ms	4.5	eq	台湾新北市
2	2015-10-19	10:17:36.1	24.94	122.01	7	Ms	5.3	eq	台湾宜兰县附近海域
3	2015-10-19	08:40:08.1	32.44	104.97	24	Ms	3.6	eq	四川省广元市青川县
4	2015-10-17	23:54:21.4	23.80	107.22	5	ML	1.4	eq	广西田东
5	2015-10-17	23:50:37.4	38.99	118.56	22	ML	2.2	eq	河北唐山
6	2015-10-17	23:44:57.8	31.02	103.31	10	ML	1.2	eq	四川汶川

另一种通用的文件格式是 `csv` 文件，上述代码使用 `read.csv()` 函数读取了 `earthquake.csv` 文件，该文件使用逗号进行分割，与 `read.csv()` 函数的默认分隔符相同，故无须指定其他参数。这份文件下载自国家地震数据中心，该网站提供最近的地震数据，`earthquake.csv` 文件包含从 2015 年 9 月 21 日到 2015 年 10 月 19 日这 4 个星期的地震数据。

使用 `head()` 函数查看 `earthquake` 中的前 6 行数据，数据框 `earthquake` 由 9 个向量组成，其中 `date` 和 `time` 分别为每一条地震数据的日期和时间，`wd` 和 `jd` 则是地震发生的纬度和经度，`address` 是地震发生的地点，其他变量则是有关地震的其他指标数据。

```
> points(earthquake$jd, earthquake$wd, col=earthquake$level, cex=0.1, pch=19)
> table(earthquake$level)
```

mb	ML	Ms
5	1699	38

上述代码中 `points()` 函数继续添加了一些坐标点。`points()` 函数的前两个参数给出了点的横、纵坐标数据，第三个参数 `col` 则表明为 `level` 不同的点绘制不同的颜色，参数 `cex` 和 `pch` 共同作用，绘出了非常小的实心圆圈。

运行结果中的点大部分都是红色，只有西藏自治区和台湾地区附近能看到一些绿色的点，对照上述代码中 `table()` 函数为 `level` 变量返回的列联表，红色的点表明地震类型为 `ML`，绿色的点表明地震类型为 `Ms`，地震类型为 `mb` 的 5 个地震点则被淹没在图中。此外，一些发生在海中的和边界国家的地震也被我国的地震探测台记录了下来。

尽管在短短 4 周时间内我国就发生了近 1 700 次地震，但这些地震绝大部分都是震级小于 3 级的弱震，只有 11 次达到了 6 级，而且震级较高的地震大部分发生在海中，因此并不会对人们的生活产生较大的影响。有关地震震级值的信息存储在 `valve` 变量中。

本节讨论了如何利用非 `shp` 格式的文件在地图中添加信息，这些不是 `shp` 格式的文件不仅在绘制地图中能起到作用，也能用于进行其他分析。

第 14 章 使用 R 构建支持向量机

本章的主题为支持向量机模型的构建，这是一个专业的、处于学科前沿的重要算法，在本章中，读者将学习支持向量机的算法思想与构建方法。本章将重点讨论如何比较不同模型的优劣，这包括优化参数及其他的一些小问题。

14.1 构建一个简单的支持向量机

本节将介绍有关支持向量机的基础知识，包括支持向量机的算法思想与构建支持向量机的方法。R 提供了专门的程序包用于构建支持向量机，本节将带领读者快速了解支持向量机模型的基础知识。

14.1.1 支持向量机的算法原理

支持向量机是一种用于分类的新兴算法，在模式识别和机器学习领域有广泛而深刻的重要应用。与因子分析和主成分分析恰好相反，支持向量机通过升维的方式解决分类问题，简单来说，就是将在低维空间中无法分类的样本点映射到高维空间中，从而将样本点分开，以达到分类的目的。

图 14.1 就是一个无法在低维空间中分类的典型例子。在图 14.1 中，红色的点和蓝色的点各自形成一个圆圈，蓝色的点恰好将红色的点完全圈了起来，很明显，在这个二维空间中我们不可能找到任何一条直线或者曲线将这两类点分开。

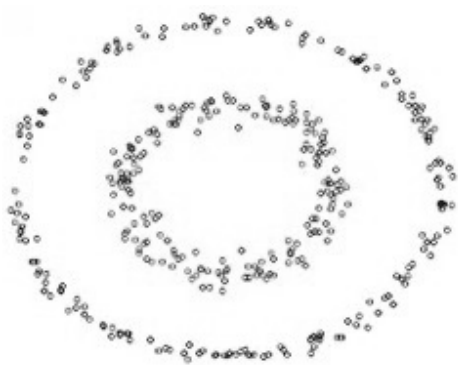


图 14.1 二维空间中的分类问题

支持向量机提供了一种解决如图 14.1 所示分类问题的思路，即将图 14.1 中位于二维空间中的点映射到三维空间中。假设图 14.1 中的点以 a 、 b 为横、纵坐标，一种可行

的映射方式为 $\begin{cases} x = a \\ y = b \\ z = a \times b \end{cases}$ ，使用该映射方式可为每一个样本点计算出一组新坐标，将它们
在三维空间中绘出来，即可得到如图 14.2 所示的三维空间分布。

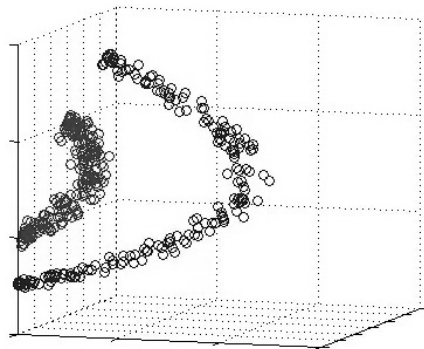


图 14.2 三维空间中的分类问题

显然，将样本点从二维空间映射到三维空间后，样本数据形成了两个坡度不同的圆圈，此时能够找到一个平面将它们分开。图 14.2 只画出了二分之一的样本数据，但容易想象，另外二分之一的样本数据也能被同一个平面分开。

除上文中给出的映射方式外，一些其他的映射方式也能起到相同的作用，比如 $\begin{cases} x = a \\ y = b \\ z = a + b \end{cases}$ 、 $\begin{cases} x = a \\ y = b \\ z = a^2 + b^2 \end{cases}$ 等。我们希望模型能达到最好的分类效果，这意味着每一类中的点都距离分类平面最远。

考虑图 14.3 所示的分类问题，图中原点和叉点代表两种不同类别的样本点，对于支持向量机来说，那些远离分类平面的点不具有研究意义，我们只关心离分类平面最近的样本点，在图 14.3 中，分类平面是一条线，离这条线最近的则是串在两条虚线上的一个圆点和两个叉点。我们称这些离分类平面最近的点为支持向量，显然，我们总是希望分类平面能够距离两类点都尽量远。

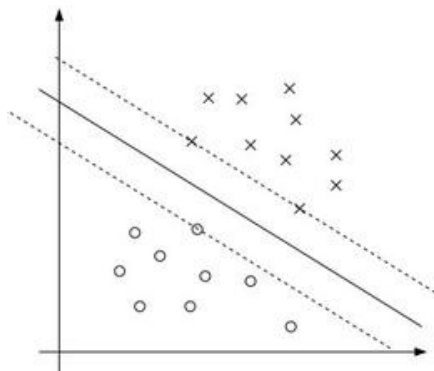


图 14.3 支持向量示意图

无论样本数据在低维空间中多么混乱不可分，支持向量机总能找到一个够高的维度将样本点完全分开，因此，支持向量机只关心支持向量而不关心远离分类平面的那些点。对于支持向量机来说，最重要的事情是找到一组合适的映射，使得被映射后的样本数据分离而且尽可能离得越远越好。

支持向量机中的映射关系也称为核函数，核函数的概念在神经网络中也被用到。在真实问题中，样本数据往往涉及成百上千个维度，核函数将包含上千个映射关系，此时找到合适的核函数是一件非常复杂、困难的事情，不过好消息是 R 提供的程序包中已经封装好了 4 种常用的核函数，并提供了一些参数用于优化核函数，它们能够解决大多数常见的问题。

14.1.2 构建一个简单的支持向量机

在正式构建支持向量机模型之前，首先需要准备一些适合支持向量机模型的问题。支持向量机最擅长的是解决非线性的分类问题，如下代码准备了一些非线性的分类样本。

```
> x <- c(runif(50,0,1),runif(100,1,3),runif(50,3,4))
> y <- runif(200,0,1)
> z <- c(rep(0,50),rep(1,100),rep(0,50))
> svm.data <- cbind(x,y,z)
```

`runif()` 函数用于生成服从均匀分布的随机数，它接受三个参数，第一个参数指定生成随机数的个数，第二个参数指定随机数的最小值，第三个参数指定随机数的最大值，后两个参数一块给出了随机数的范围，它们的默认值分别为 0 和 1。

上述代码中第 1 行命令生成了变量 `x`，它含有 200 个数据，前 50 个数据是分布在 0~1 之间的随机数，中间的 100 个数据是分布在 1~3 之间的随机数，后 50 个数据是分布在 3~4 之间的随机数。第 2 行命令生成了变量 `y`，它含有 200 个分布在 0~1 之间的均匀随机数。第 3 行命令给出了类别标签 `z`，它包含 100 个 0 及 100 个 1，其中前 50 个数据和后 50 个数据属于类别 0，中间的 100 个数据属于类别 1，`cbind()` 函数将 `x`、`y`、`z` 按行联合为一个数据框。

```
> plot(x,y,col=c(rep("red",50),rep("blue",100),rep("red",50)))
```

为了能直观地了解样本数据的分布情况，上述代码使用 `plot()` 函数绘出了 `svm.data` 中的数据点。在上述代码中，`x` 和 `y` 分别为图形的横轴和纵轴，参数 `col` 则给出了每一个点的颜色，我们为参数 `col` 指定了一个长度为 200 的颜色向量，使得前 50 个数据和后 50 个数据为红色，中间 100 个数据为蓝色，这与变量 `z` 给出的类别标签相吻合。

图 14.4 是 R 的最终结果。观察图 14.4，样本点毫无规律地均匀散落在二维平面上，处于左、右两端的数据呈红色，处于中间的数据呈蓝色，显然，不可能找到任何一条直线将两种类别的数据完全分开。

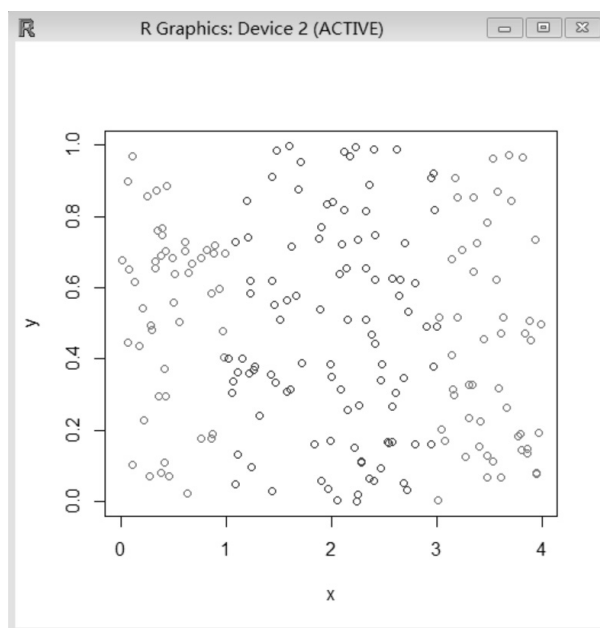


图 14.4 非线性分类的样本数据

```
> library(e1071)
> svm.fit <- svm(z~x+y,data=svm.data)
> summary(svm.fit)

Call:
svm(formula = z ~ x + y, data = svm.data)

Parameters:
  SVM-Type:  eps-regression
SVM-Kernel:  radial
    cost:    1
   gamma:   0.5
  epsilon:   0.1

Number of Support Vectors: 147
```

程序包 `e1071` 提供了有关构建支持向量机模型的函数，上述代码首先加载了包 `e1071`，然后调用了 `svm()` 函数以构建支持向量机模型。`svm()` 函数的参数输入格式与 `lm()` 函数、`glm()` 函数等函数都十分相似，在上述代码中，第一个参数指定 z 为类别标签， x 和 y 为提供分类信息的变量，参数 `data` 则表明从 `svm.data` 中抽取这些变量。

`summary()` 函数同样能够查看 `svm` 模型的基本信息。观察上述代码中第 3 行命令返回的结果，R 给出了模型的公式及一些主要参数，`SVM-Kernel` 表明此处使用的核函数是 `radial` 核函数，即径向基核函数，这是一种基于样本点之间的距离决定映射方式的函数。

Number of Support Vectors 则表明在这个模型中支持向量的数目为 147，考虑到我们仅有 200 个数据，显然，分类平面距离大部分的样本点都相当近。

```
> head(predict(svm.fit))
      1      2      3      4      5      6
-0.007537115  0.050676140  0.261376210  0.247945340 -0.020877559
0.050299113
> svm.pre <- ifelse(predict(svm.fit)>0,1,0)
> head(svm.pre)
 1 2 3 4 5 6
0 1 1 1 0 1
```

summary() 函数并不能给出有关模型更详细的信息。对于分类模型来说，我们最关心的是模型的准确度，即模型是否能正确地为本点分类。predict() 函数能根据模型进行预测，上述代码中第 1 行命令查看了模型 svm.fit 对 200 个样本点的前 6 个预测结果，预测结果围绕 0 上下波动，它度量了每一个样本点属于类别 1 的可能性，该值为正时，说明样本点属于类别 1，该值为负时，说明样本点属于类别 0，该值的绝对值越大，说明预测结果越有可能是正确的。

上述代码中第 2 行命令使用 ifelse() 语句创建了另一个长度为 200 的变量 svm.pre，当 predict(svm.fit) 大于 0 时，对应的 svm.pre 的值就为 1，否则就为 0。显然，svm.pre 存储了所有 svm.fit 对样本数据的预测结果。

```
> n <- ifelse(svm.pre==z,1,0)
> sum(n)
[1] 138
```

上述代码再次调用了 ifelse() 函数用于创建向量 n ，当 svm.pre 的值与 z 相同时， n 的值就为 1，否则， n 的值就为 0。即对于预测结果正确的样本点来说，对应的 n 值为 1；对于预测结果错误的样本来说， n 的值就为 0。 n 同样是一个长度为 200 的变量，sum() 函数求出了 n 的和，其结果为 138，显然，svm.fit 模型预测正确的样本数目为 138，我们只有 200 个样本，因此模型的准确度达到了 69%。

```
> col <- ifelse(svm.pre==0,"red","blue")
> plot(x,y,col=col)
```

为预测结果绘出图形能够直观了解到模型的优劣。上述代码中第 1 行命令将所有预测类别为 0 的点记录为 “red”，将所有预测类别为 1 的点记录为 “blue”，并将这 200 条颜色信息存储在 col 中。第 2 行命令调用 plot() 函数绘出了散点图，并使用 col 作为散点的颜色参数。其最终结果如图 14.5 所示。

观察图 14.5，svm.fit 将大部分点都预测为类别 1，只有处于两端的一些散点被预测为类别 0。模型 svm.fit 很好地捕捉到样本数据中存在的带状规律，将样本数据分为三个带状，这种非线性规律正是其他分类方法难以捕捉到的。

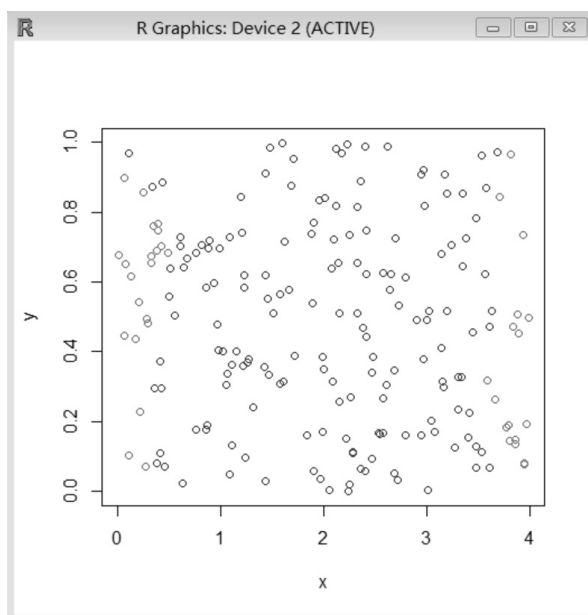


图 14.5 使用径向基核函数预测样本数据的类别

14.1.3 使用其他核函数构建支持向量机

e1071 包提供的 `svm()` 函数包含多种核函数，除径向基核函数外，线性核函数、多项式核函数、S 型核函数是可供选择的三种核函数。与径向基核函数相比，它们各有优劣。

```
> svm.line <- svm(z~x+y,data=svm.data,kernel="linear")
> svm.linepre <- ifelse(predict(svm.line)>0,1,0)
> n <- ifelse(svm.linepre==z,1,0)
> sum(n)
[1] 100
> col <- ifelse(svm.linepre==0,"red","blue")
> plot(x,y,col=col)
```

上述代码在 `svm()` 函数中增加了参数 `kernel`，它用于指定支持向量机的核函数，此处参数 `kernel` 指定为 `linear`，即使用线性核函数构建模型。接下来的三行代码同样使用 `predict()` 函数为样本数据预测了类别，并统计出所有预测正确的样本点的个数。

观察第 4 行命令的返回结果，线性核函数仅正确预测出 100 个样本点，正确率只有 50%。接下来的两行命令同样按照预测的类别绘出了散点图，预测类别为 0 的样本为红色，预测类别为 1 的样本为蓝色。图 14.6 为 R 的返回结果，观察图 14.6，使用线性核函数的支持向量机将样本点全部预测为类别 1，这显然不是一个好的预测结果。

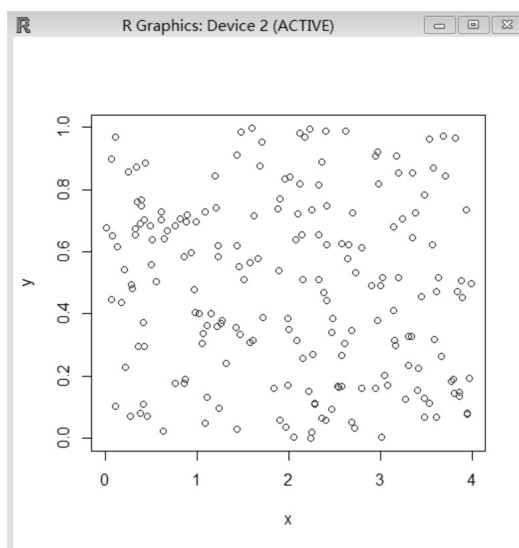


图 14.6 使用线性核函数预测样本数据的类别

```
> svm.poly <- svm(z~x+y,data=svm.data,kernel="polynomial")
> svm.polypre <- ifelse(predict(svm.poly)>0,1,0)
> n <- ifelse(svm.polypre==z,1,0)
> sum(n)
[1] 100
> col <- ifelse(svm.polypre==0,"red","blue")
> plot(x,y,col=col)
```

上述代码修改了参数 `kernel`，将其指定为 `polynomial`，即使用多项式核函数构建支持向量机，它的预测准确率同样为 50%，观察图 14.7，它同样全部由蓝色样本点构成，显然，多项式核函数构建的支持向量机同样认为所有的样本数据类别都为 1。

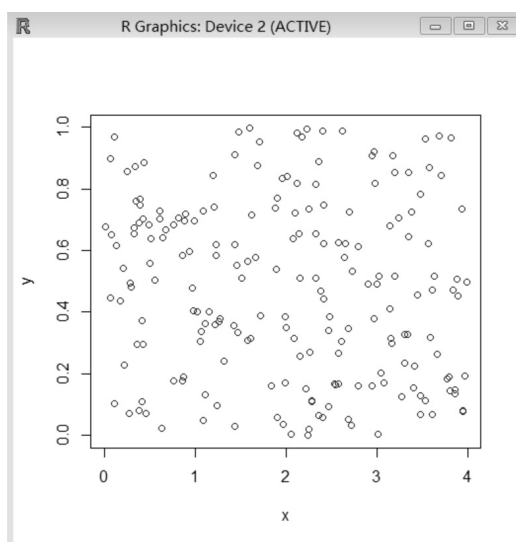


图 14.7 使用多项式核函数预测样本数据的类别

线性核函数和多项式核函数都擅长解决线性分类问题，径向基核函数则擅长解决非线性的分类问题。考虑上述三个支持向量机模型的准确率，我们构造的带状分类问题显然具有非线性的决策边界，因此线性核函数和多项式核函数都表现一般。

```
> svm.sig <- svm(z~x+y,data=svm.data,kernel="sigmoid")
> svm.sigpre <- ifelse(predict(svm.sig)>0,1,0)
> n <- ifelse(svm.sigpre==z,1,0)
> sum(n)
[1] 113
> col <- ifelse(svm.sigpre==0,"red","blue")
> plot(x,y,col=col)
```

最后一种可选的核函数是 S 型核函数，上述代码中指定参数 `kernel` 的值为 `sigmoid`，即使用 S 型核函数构建支持向量机模型，该模型正确预测的样本数目为 113，其正确率为 56.5%，好于线性核函数和多项式核函数，但比径向基核函数更差。

根据 S 型核函数的预测结果同样能够绘出散点图 14.8。观察图 14.8，样本的预测类别呈现非常奇怪的分布，它们被分为几个倾斜的带状，蓝色样本和红色样本交错分布。显然，S 型样本对样本数据呈现的带状规律过分敏感，因此出现了图 14.8 所示的情况。

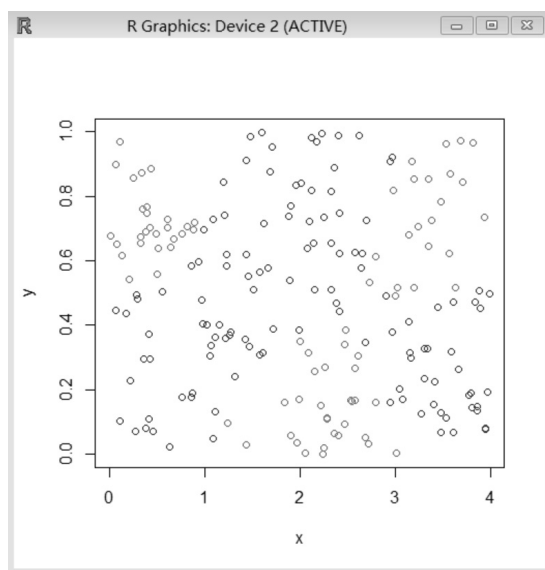


图 14.8 使用 S 型核函数预测样本数据的类别

在选择核函数时，我们通常倾向于选择径向基核函数，线性核函数可以看作径向基核函数的一种特例，S 型核函数又比线性核函数需要更多的参数。因此，径向基核函数总是支持向量机的首选，实际上，它也是神经网络核函数的首选。

14.2 优化支持向量机的参数

在 14.1 节中初步构建了 4 种核函数不同的支持向量机，本节将进一步讨论如何优化

这些模型。本节的讨论范围包括参数 `degree`、参数 `cost` 和参数 `gamma` 的优化，读者也将对每种核函数有更深入的认识。

14.2.1 优化参数 `degree`

参数 `degree` 是一种仅存在于多项式核函数中的参数，它代表多项式核函数的次数，多项式次数的概念在第 10 章中有关多项式回归的部分中提到过。一种容易产生的猜想为：多项式核函数的次数越高，映射关系就越复杂，将不同类别的点分开的概率也就越高。为了验证这种猜想是否正确，不妨设置几个不同的 `degree` 参数水平，并比较不同模型的准确度。

```
> svm.poly3 <- svm(z~x+y,data=svm.data,kernel="polynomial",degree=3)
> svm.poly3pre <- ifelse(predict(svm.poly3)>0,1,0)
> n3 <- ifelse(svm.poly3pre==z,1,0)
> sum(n3)
[1] 100
```

上述代码中第 1 行命令指定参数 `kernel` 为 `polynomial`，参数 `degree` 为 3，构建了 `degree` 水平为 3 的多项式核函数支持向量机。第 2 行命令使用 `predict()` 函数预测了样本点的类别，并将预测结果记录在 `svm.poly3pre` 中。第 3、4 行命令比较了预测结果和真实结果的差异，并统计了所有预测正确的数目。R 的返回结果为 100，显然，`degree` 为 3 时模型仅能正确预测出一半的样本点类别。

```
> svm.poly6 <- svm(z~x+y,data=svm.data,kernel="polynomial",degree=6)
> svm.poly6pre <- ifelse(predict(svm.poly6)>0,1,0)
> n6 <- ifelse(svm.poly6pre==z,1,0)
> sum(n6)
[1] 119
> svm.poly9 <- svm(z~x+y,data=svm.data,kernel="polynomial",degree=9)
> svm.poly9pre <- ifelse(predict(svm.poly9)>0,1,0)
> n9 <- ifelse(svm.poly9pre==z,1,0)
> sum(n9)
[1] 104
> svm.poly12 <- svm(z~x+y,data=svm.data,kernel="polynomial",degree=12)
> svm.poly12pre <- ifelse(predict(svm.poly12)>0,1,0)
> n12 <- ifelse(svm.poly12pre==z,1,0)
> sum(n12)
[1] 113
```

上述代码将构建 `svm.poly3` 模型的代码重复了三次，分别构建了 `degree` 水平为 6、9、12 的多项式核函数支持向量机，并统计了这三个模型的预测结果准确度。

当 `degree` 水平为 6 时，模型能够准确预测出 119 个样本点的类别，这一数据比模型的 `degree` 水平为 3 时要好，但当 `degree` 水平为 9 时，模型能够准确预测类别的样本点又变为了 104，准确度要差于 `degree` 水平为 6 的模型。而当 `degree` 水平为 12 时，类别预测正确的样本点数目再次上升为 113。为了更直观地观察到模型预测结果的变化，不妨

为 4 个模型逐一绘出散点图。

```
> par(mfrow=c(2,2))
> col <- ifelse(svm.poly3pre==0,"red","blue")
> plot(x,y,col=col,main="degree=3")
> col <- ifelse(svm.poly6pre==0,"red","blue")
> plot(x,y,col=col,main="degree=6")
> col <- ifelse(svm.poly9pre==0,"red","blue")
> plot(x,y,col=col,main="degree=9")
> col <- ifelse(svm.poly12pre==0,"red","blue")
> plot(x,y,col=col,main="degree=12")
```

上述代码绘出了 4 张 degree 水平不同的散点图。第 1 行命令创建了一个两行两列的画布准备放置这 4 张散点图，第 3、4 行代码创建了 col 变量，并根据 col 给出的颜色信息绘出了标题为 degree=3 的散点图，其他三张散点图的绘制方法相类似。

观察图 14.9，其中蓝色的样本点代表样本数据的预测类别为 1，红色的样本点代表样本数据的预测类别为 0。当 degree 为 3 时，模型将全部样本点都预测为类别 1，当 degree 为 6、9、12 时，分别在散点图的两侧出现了一些红色样本点，表明有一些样本居于两侧的样本点被正确预测，这种预测结果与 14.1 节中径向基核函数的预测结果颇有相似之处，它们同样捕捉到了样本数据的带状分布，同样不能正确预测两类数据相交处的样本点类别。

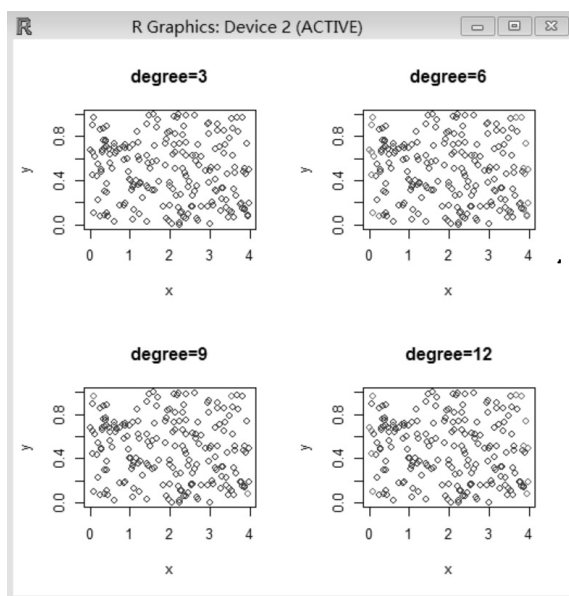


图 14.9 degree 水平不同的 4 张散点图

```
> for(i in 1:20){
+ svm.poly <- svm(z~x+y,data=svm.data,kernel="polynomial",degree=i)
+ svm.polypre <- ifelse(predict(svm.poly)>0,1,0)
+ n <- ifelse(svm.polypre==z,1,0)
+ print(sum(n))
}
```

```
+ }  
[1] 100  
[1] 129  
[1] 100  
[1] 125  
[1] 101  
[1] 119  
[1] 102  
[1] 117  
[1] 104  
[1] 114  
[1] 107  
[1] 113  
[1] 105  
[1] 113  
[1] 107  
  
WARNING: reaching max number of iterations  
[1] 114  
  
WARNING: reaching max number of iterations  
[1] 112  
  
WARNING: reaching max number of iterations  
[1] 112  
  
WARNING: reaching max number of iterations  
[1] 111  
  
WARNING: reaching max number of iterations  
[1] 112
```

为了找出模型准确率随 `degree` 参数变化而变化的规律，上述代码使用 `for` 循环逐一返回了 `degree` 取值为 1~20 之间值时模型能准确预测出的样本类别数目。当 `degree` 水平为奇数时，模型能够预测正确的样本数目为 100、100、101、102、104、107、105、107、112、111；当 `degree` 水平为偶数时，模型能够预测正确的样本数目为 129、125、119、117、114、113、114、112、112。

显然，当 `degree` 水平为奇数时，`degree` 越大，模型表现越好；当 `degree` 水平为偶数时，`degree` 越大，模型表现越差，而当 `degree` 水平足够大时，模型的准确度开始趋于平稳。这说明多项式核函数的最佳次数与样本数据的内部结构有很大关系，在本例中，`degree` 为 2 时模型有最佳的预测结果，准确度能达到 64.5%。

此外，当 `degree` 高于 15 时，R 返回了警告信息，表明此时给出的次数太高，过高的次数将带来过大的计算量，我们的样本数据量非常小，因此 R 运行了几十秒便返回了结果，但容易理解，当样本数据量较大时，设置过高的 `degree` 将非常危险。

14.2.2 优化参数 cost

参数 `cost` 是一种可以和 4 种核函数中任何一种相搭配的参数，它是一个正则化的参数，用于给出惩罚因子，即支持向量机在优化模型时对导致模型预测效果变差的因素的惩罚力度。参数 `cost` 的默认取值为 1，也可以取为更高的值。

```
> svm.cost1 <- svm(z~x+y,data=svm.data,cost=1)
> svm.cost1pre <- ifelse(predict(svm.cost1)>0,1,0)
> n1 <- ifelse(svm.cost1pre==z,1,0)
> sum(n1)
[1] 138
> svm.cost2 <- svm(z~x+y,data=svm.data,cost=2)
> svm.cost2pre <- ifelse(predict(svm.cost2)>0,1,0)
> n2 <- ifelse(svm.cost2pre==z,1,0)
> sum(n2)
[1] 141
> svm.cost3 <- svm(z~x+y,data=svm.data,cost=3)
> svm.cost3pre <- ifelse(predict(svm.cost3)>0,1,0)
> n3 <- ifelse(svm.cost3pre==z,1,0)
> sum(n3)
[1] 142
> svm.cost4 <- svm(z~x+y,data=svm.data,cost=4)
> svm.cost4pre <- ifelse(predict(svm.cost4)>0,1,0)
> n4 <- ifelse(svm.cost4pre==z,1,0)
> sum(n4)
[1] 145
```

上述代码一共构建了 4 个支持向量机模型，在 `svm()` 函数中并未指定参数 `kernel`，故这 4 个模型都使用默认的径向基核函数来构建模型。在这 4 个模型中 `cost` 参数分别取值为 1、2、3、4，`svm.cost1pre`、`svm.cost2pre`、`svm.cost3pre`、`svm.cost4pre` 中存储了每个模型的预测类别，`sum()` 函数则统计出每个模型类别预测正确的样本数目。

这 4 个模型能够正确预测的样本数目分别为 138、141、142 和 145，显然，随着 `cost` 水平的增大，模型的表现越来越好，在 14.1 节中我们已经发现使用径向基核函数构建的支持向量机能够很好地捕捉到样本数据的带状规律，不妨绘出散点图，以便查看改变 `cost` 的水平时，模型是否仍能捕捉到带状规律。

```
> par(mfrow=c(2,2))
> col <- ifelse(svm.cost1pre==0,"red","blue")
> plot(x,y,col=col,main="cost=1")
> col <- ifelse(svm.cost2pre==0,"red","blue")
> plot(x,y,col=col,main="cost=2")
> col <- ifelse(svm.cost3pre==0,"red","blue")
> plot(x,y,col=col,main="cost=3")
> col <- ifelse(svm.cost4pre==0,"red","blue")
> plot(x,y,col=col,main="cost=4")
```

上述代码同样绘出了一张两行两列的组合图形，它包含 4 张小的散点图，每张散点图都用蓝色代表样本点预测类别为 1，用红色代表样本点预测类别为 0。观察图 14.10，这 4 张散点图，每张图片的红色样本点都分散在两侧，呈现出较好的带状规律。模型的准确度告诉我们随着 `cost` 的增大，模型也越来越准确，但观察这 4 张散点图，样本分布并没有发生显著的改变，显然，修改 `cost` 对模型的优化作用是细微的。即便如此，我们仍希望找到一个最佳的 `cost` 值，使模型的准确度达到最高。

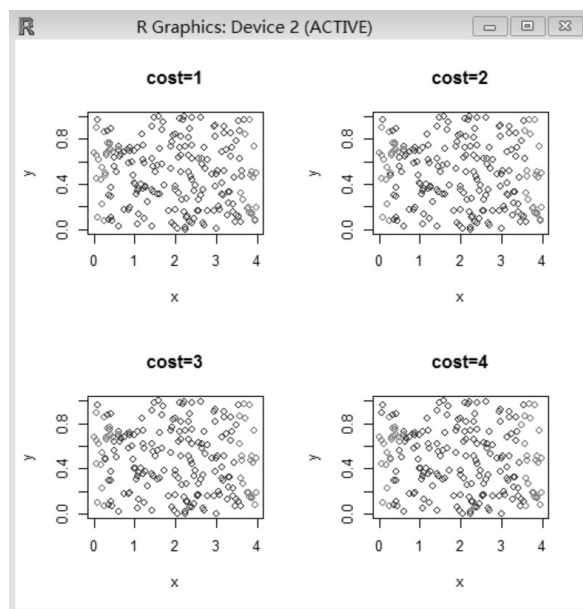


图 14.10 `cost` 水平不同的 4 张散点图

```
> for(i in 1:20){
+ svm.cost <- svm(z~x+y,data=svm.data,cost=i)
+ svm.costpre <- ifelse(predict(svm.cost)>0,1,0)
+ n <- ifelse(svm.costpre==z,1,0)
+ print(sum(n))
+ }
[1] 138
[1] 141
[1] 142
[1] 145
[1] 144
[1] 144
[1] 141
[1] 140
[1] 140
[1] 141
[1] 140
[1] 138
[1] 138
```

```
[1] 140
[1] 143
[1] 144
[1] 144
[1] 144
[1] 144
[1] 144
```

上述代码查看了 `cost` 取值为 1~20 时模型能够准确预测出的样本数目。这 20 个模型仍然使用径向基核函数。观察 R 的返回结果，正确预测出的样本数目首先从 138 上升至 145，然后逐渐下降到 138，再次回升至 144，并稳定下来。显然，当 `cost` 取值为 4 时，模型的准确度达到 72.5%，此时模型是最准确的。

参数 `cost` 和参数 `degree` 经常配合使用，我们希望找到最佳的参数组合，使模型表现最好，容易理解，研究两个参数构成的最佳组合要比单个单个地研究参数麻烦，但循环函数能帮助我们较容易地实现优化目标。此外，在研究参数 `cost` 和参数 `degree` 时，我们仅仅简单地在原始数据集上验证了模型，使用交叉验证法验证模型将得到更准确的结果。

14.2.3 优化参数 gamma

参数 `gamma` 是我们关心的最后一个参数，它能够与除线性核函数外的其他三种核函数相配合。参数 `gamma` 的默认值为 0.5，不妨考察当它取整数值时模型的准确度。

```
> svm.gamma1 <- svm(z~x+y,data=svm.data,gamma=1)
> svm.gamma1pre <- ifelse(predict(svm.gamma1)>0,1,0)
> n1 <- ifelse(svm.gamma1pre==z,1,0)
> sum(n1)
[1] 138
> svm.gamma2 <- svm(z~x+y,data=svm.data,gamma=2)
> svm.gamma2pre <- ifelse(predict(svm.gamma2)>0,1,0)
> n2 <- ifelse(svm.gamma2pre==z,1,0)
> sum(n2)
[1] 142
> svm.gamma3 <- svm(z~x+y,data=svm.data,gamma=3)
> svm.gamma3pre <- ifelse(predict(svm.gamma3)>0,1,0)
> n3 <- ifelse(svm.gamma3pre==z,1,0)
> sum(n3)
[1] 146
> svm.gamma4 <- svm(z~x+y,data=svm.data,gamma=4)
> svm.gamma4pre <- ifelse(predict(svm.gamma4)>0,1,0)
> n4 <- ifelse(svm.gamma4pre==z,1,0)
> sum(n4)
[1] 135
```

上述代码使用径向基核函数构建了 4 个支持向量机模型，它们的 `gamma` 值分别取为 1、2、3、4，并用与优化参数 `degree`、参数 `cost` 类似的代码统计了每个模型能够正确

预测类别的样本数目。当 γ 值分别取为 1、2、3、4 时，模型能够正确预测类别的样本数目分别为 138、142、146、135，随着参数 γ 值的增加，支持向量机的准确度也会增加。

```
> par(mfrow=c(2,2))
> col <- ifelse(svm.gamma1pre==0,"red","blue")
> plot(x,y,col=col,main="gamma=1")
> col <- ifelse(svm.gamma2pre==0,"red","blue")
> plot(x,y,col=col,main="gamma=2")
> col <- ifelse(svm.gamma3pre==0,"red","blue")
> plot(x,y,col=col,main="gamma=3")
> col <- ifelse(svm.gamma4pre==0,"red","blue")
> plot(x,y,col=col,main="gamma=4")
```

上述代码绘出了一张两行两列的组合图形，它根据 4 个模型的预测结果绘出了 4 张散点图，其中蓝色代表预测类别为 1，红色代表预测类别为 0。观察图 14.11，当 γ 值增大时，预测结果为 0 的样本点从两侧向中间平移，但总体来说，预测结果为 0 的样本点仍然是少数。

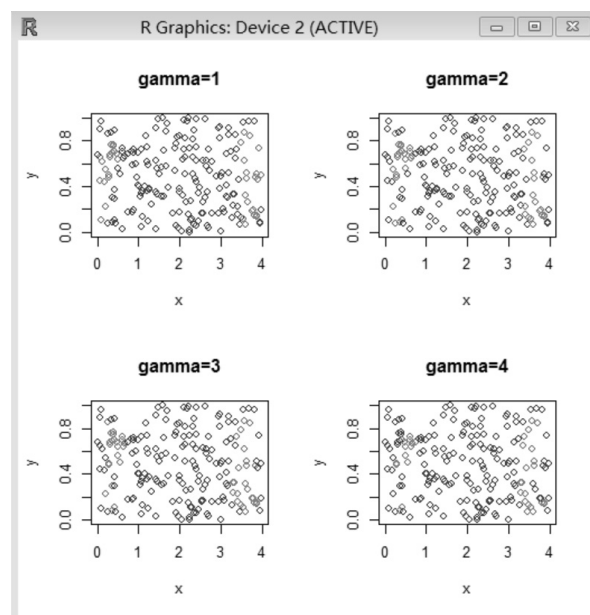


图 14.11 γ 水平不同的四张散点图

```
> for(i in 1:20){
+ svm.gamma <- svm(z~x+y,data=svm.data,gamma=i)
+ svm.gammapre <- ifelse(predict(svm.gamma)>0,1,0)
+ n <- ifelse(svm.gammapre==z,1,0)
+ print(sum(n))
+ }
[1] 138
[1] 142
```

```

[1] 146
[1] 135
[1] 133
[1] 130
[1] 124
[1] 116
[1] 111
[1] 111
[1] 111
[1] 111
[1] 112
[1] 109
[1] 108
[1] 106
[1] 106
[1] 106
[1] 105
[1] 105

```

上述代码使用 for 循环查看了当 `gamma` 取值为 1~20 时构建的 20 个模型中预测正确的样本数目。观察 R 的返回结果，模型的准确度首先随着 `gamma` 值的增加而有一个短暂的增加，并在 `gamma` 取值为 3 时达到顶点，但当 `gamma` 值继续增大后，模型的准确度便一路下滑，显然，在本例中，样本数据并不需要较大的 `gamma` 水平。

```

> a <- c()
> for(i in 1:10){
+ for(j in 1:10){
+ for(k in 1:10){
+ svm.fit <- svm(z~x+y,data=svm.data,degree=i,cost=j,gamma=k)
+ svm.pre <- ifelse(predict(svm.fit)>0,1,0)
+ n <- ifelse(svm.pre==z,1,0)
+ result <- c(i,j,k,sum(n))
+ a <- rbind(a,result)
+ }}}
> head(a)
      [,1] [,2] [,3] [,4]
result   1    1    1 138
result   1    1    2 142
result   1    1    3 146
result   1    1    4 135
result   1    1    5 133
result   1    1    6 130

```

一种筛选最佳参数组合的办法是遍历所有参数组合，并从中筛选出最佳的参数组合。上述代码首先创建了一个空向量 `a`，我们打算向其中放入最后的结果。接下来是一个三重嵌套 for 循环语句，在这个嵌套循环中，`i` 为 `degree` 的取值范围，`j` 为 `cost` 的取值范围，`k` 为 `gamma` 的取值范围。这三个范围均为 1~10，显然，循环体将循环 1 000 次。

循环体中首先构建了支持向量机模型，并统计了每一个模型能够正确预测的样本数目，向量 `result` 存储了每一次循环对应的 i 值、 j 值和 k 值，以及该循环预测正确的样本数目，`rbind()` 函数将 `result` 按行缀连了起来。

查看变量 a 的前 6 个元素，显然，它存储了我们想要的结果，在这 6 个元素中， i 值和 j 值都为 1， k 值则从 1 增加到 6，第 4 列对应这 6 个模型正确分类的样本数目，我们感兴趣的是第 4 列中最大的元素。

```
> a[which(a[,4]==max(a[,4])),]
      [,1] [,2] [,3] [,4]
result  1    3    2  153
result  2    3    2  153
result  3    3    2  153
result  4    3    2  153
result  5    3    2  153
result  6    3    2  153
result  7    3    2  153
result  8    3    2  153
result  9    3    2  153
result 10    3    2  153
```

上述代码在查看 a 中数据时给出了一个筛选条件，`which()` 函数筛选出 a 中第 4 列最大的元素。观察 R 的返回结果， a 中第 4 列最大的元素为 153，它们对应的 j 值均为 3， k 值均为 2，而 i 值则从 1 变化到 10。显然，在使用径向基核函数构建支持向量机时，参数 `degree` 的变化并不对模型结果产生任何影响。

根据上述代码的返回结果，在我们使用的样本数据中，当参数 `cost` 取值为 3、参数 `gamma` 取值为 2 时，模型能正确预测出 153 个样本数据的类别，准确率为 76.5%，达到最高。单独优化 `cost` 时，`cost` 取 4 时为最佳模型；单独优化 `gamma` 时，`gamma` 取 3 时为最佳模型，显然，两个参数组合起来时，最优取值与单独优化参数时结果并不相同。

14.3 比较支持向量机与 Logistic 回归的优劣

本书的 10.2 节讲解了 Logistic 线性回归模型的构建方法。Logistic 线性回归模型是一种专门用于为线性数据分类的模型。本节将讨论如何在本章提供的数据集上构建 Logistic 回归模型，并对比 Logistic 回归模型分类结果与支持向量机分类结果的优劣。

```
> library(foreign)
> lm.log <- glm(z~x+y,family=binomial,data=svm.data)
> summary(lm.log)

Call:
glm(formula = z ~ x + y, family = binomial, data = svm.data)

Deviance Residuals:
      Min       1Q   Median       3Q      Max
```

```

-1.20807 -1.17270 0.00735 1.17789 1.19391

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.03624    0.39148  -0.093   0.926
x             0.02821    0.12256   0.230   0.818
y            -0.04284    0.50040  -0.086   0.932

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 277.26  on 199  degrees of freedom
Residual deviance: 277.19  on 197  degrees of freedom
AIC: 283.19

Number of Fisher Scoring iterations: 3

```

上述代码首先加载了程序包 `foreign`，第 2 行命令调用了 `foreign` 包中的 `glm()` 函数用于构建 Logistic 回归模型，该函数的使用方法与 `svm()` 函数十分相似，在上述命令中第一个参数表明 z 为分类标签， x 和 y 为用于分类的变量，第二个参数 `family` 表明使用 `binomial` 作为连接函数，即构建 Logistic 分类模型，第三个参数则给出了构建模型的变量来源。

`summary()` 函数查看了 Logistic 模型的摘要，其中，`Coefficients` 元素给出了常数项和变量 x 、变量 y 的系数与 p 值，它们的系数绝对值都非常小，表明 Logistic 模型并未找到能对样本分类起到明显作用的变量，此外，它们的 p 值都远远大于 0.05，表明这三个系数都不可信。

```

> anova(lm.log, test="Chisq")
Analysis of Deviance Table

Model: binomial, link: logit

Response: z

Terms added sequentially (first to last)

      Df Deviance Resid. Df Resid. Dev Pr(>Chi)
NULL                                199      277.26
x      1  0.058973      198      277.20  0.8081
y      1  0.007328      197      277.19  0.9318

```

为了排除变量之间的相互影响，上述代码使用 `anova()` 函数对 Logistic 回归模型进行了似然比卡方检验，观察 R 的返回结果，变量 x 和变量 y 能够产生的模型偏差均小于 0.1，是非常非常小的。 p 值则大于 0.05，表明这两个变量对模型的影响是不显著的。

综合考虑模型的摘要与似然比卡方检验的结果，为 `svm.data` 构建的 Logistic 线性回归模型表现显然很糟糕。这意味着 `svm.data` 的分类边界很可能不是线性的。我们仍对该

Logistic 回归模型的最终预测结果感兴趣，不妨看一看 Logistic 模型在这里给出了怎样的结果。

```
> lm.pre <- ifelse(predict(lm.log)>0,1,0)
> n <- ifelse(lm.pre==z,1,0)
> sum(n)
[1] 99
```

Logistic 回归模型为每一个样本给出的预测结果同样是一个概率的形式，上述代码使用 `predict()` 函数逐一预测了每一样本点的类别，当预测结果大于 0 时，即认为样本属于类别 1；当预测结果小于等于 0 时，即认为样本属于类别 0。第 2 行命令逐一比较了预测结果与真实结果的差异，当二者相同时，记录为 1，当二者不同时，记录为 0。这两组 `ifelse()` 语句都是在一个长度为 200 的向量上执行，因此返回的也是一个长度为 200 的向量。

`sum()` 函数统计了所有预测正确的样本个数，R 的返回结果为 99，即 Logistic 回归模型仅能正确预测出 99 个样本点的类别，这一准确率甚至没有达到 50%，也就是说，即便我们不使用任何模型随意猜测，准确率都会好于该模型。

```
> col <- ifelse(lm.pre==0,"red","blue")
> plot(x,y,col=col)
```

上述代码绘出了样本数据的预测结果，其中，红色样本点为预测类别为 0 的样本点；蓝色样本点为预测类别为 1 的样本点。观察图 14.12，红色点与蓝色点分散在左、右两侧，各占一半的空间。这两类点的交界处为一条斜线，这条斜线将图 14.12 给出的长方形切分为两个梯形。

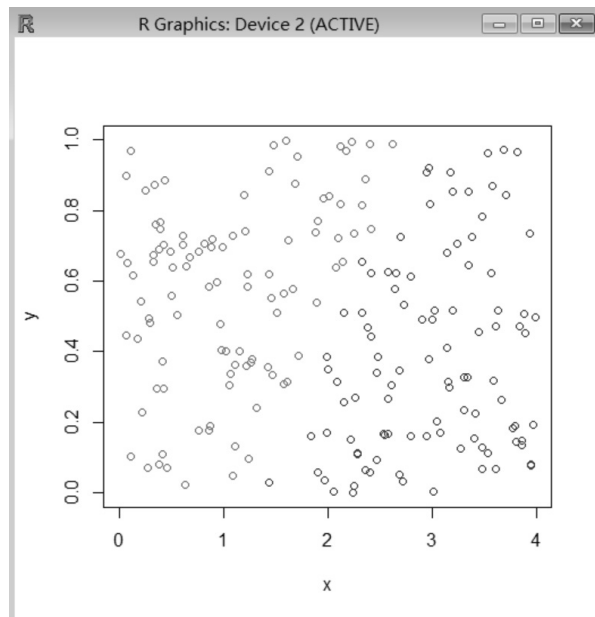


图 14.12 使用 Logistic 回归模型预测样本类别

Logistic 模型给出了一条线性的决策边界，将一半类别为 1 的样本点预测为类别 0，将一半类别为 0 的样本点预测为类别 1。它也没能正确地捕捉到样本数据中存在的带状分布规律，预测结果无法令人满意。

14.4 比较支持向量机和 KNN 聚类算法的优劣

KNN 聚类算法是另一类擅长解决非线性分类问题的算法，本节将在相同的数据集上构建 KNN 聚类算法，并对比 KNN 聚类算法与支持向量机的预测结果。

```
> library(class)
> kn <- knn.cv(svm.data[,1:2],cl=svm.data[,3],k=10)
> head(kn)
[1] 0 0 0 0 0 0
Levels: 0 1
```

上述代码首先加载了 `class` 程序包，然后调用了 `class` 包中的 `knn.cv()` 函数以构建 KNN 聚类模型，在第 2 行命令中，`knn.cv()` 函数的第一个参数表明 `svm.data` 的前两列变量为用于分类的变量，第二个参数 `cl` 则表明 `svm.data` 的第 3 列变量为分类标签，第三个参数 `k` 给出了用于计算距离的最近邻点。考虑到仅有 200 个样本数据，在这里指定 `k` 为 10。

`knn.cv()` 函数将直接返回每一样本点的预测类别，`head()` 函数查看了 `kn` 的前 6 个元素，显然，这个模型将前 6 个样本点均预测为类别 0。

```
> n <- ifelse(kn==z,1,0)
> sum(n)
[1] 191
```

直接对比 `kn` 中元素与 `z` 中元素的差异即可统计出 KNN 聚类算法的预测准确度。上述代码使用 `ifelse` 语句对比了每一样本点的预测结果与真实结果，当此二者相同时，`n` 将存入一个 1；当此二者不同时，`n` 将存入一个 0。`sum()` 函数统计了 KNN 算法能够准确预测类别的样本个数。R 的返回结果为 191，显然，KNN 聚类模型的准确率高达 85.5%，这比采用了最优参数组合的支持向量机模型还要准确。

```
> col <- ifelse(kn==0,"red","blue")
> plot(x,y,col=col)
```

上述代码同样根据预测类别的不同绘出了不同颜色的样本点，其中，预测类别为 0 的样本点为红色，预测类别为 1 的样本点为蓝色，R 的返回结果如图 14.13 所示。观察图 14.13，样本的预测类别与真实类别几乎完全一致。KNN 聚类模型堪称完美地找出了样本数据中的带状规律，并且也并未像支持向量机一样将分类线向两侧平移。

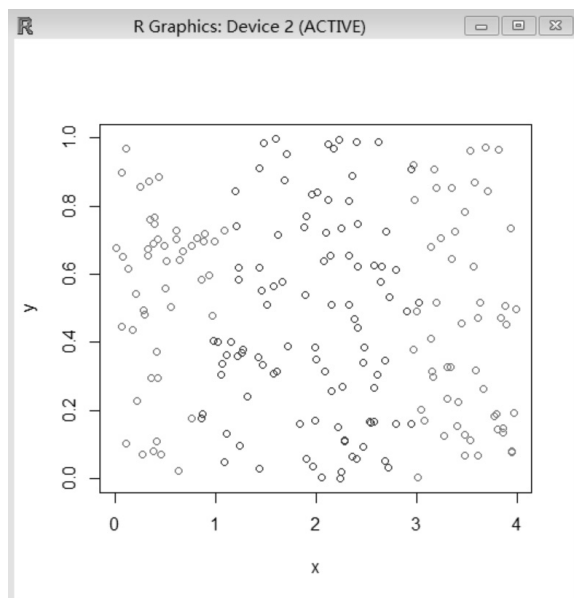


图 14.13 使用 KNN 聚类模型预测样本类别

```
> table(z, kn)
      kn
z      0  1
0     96  4
1      5 95
```

为了查看 KNN 聚类模型的更多细节，上述代码使用 `table()` 函数查看了 `kn` 与 `z` 的列联表，观察 R 的返回结果，KNN 聚类模型将 4 个真实类别为 0 的样本点预测为类别 1，将 5 个真实类别为 1 的样本点预测为类别 0，这些点显然是那些位于分界线附近的点。

在本例中，KNN 算法模型明显优于支持向量机模型，这是因为样本数据具有非常明显的非线性结构，也恰好能分成很规整的几大块。支持向量机具有 4 种核函数，同时能够处理线性的与非线性的数据结构，在一些数据结构并不具有特别明显的规律的数据集上，支持向量机将表现出最佳的分类结果。

支持向量机也擅长处理维度很高的稀疏数据，当数据过于稀疏时，KNN 算法模型不能较好地找到类别的中心点，而支持向量机则不受该约束，支持向量机也具有较快的运算速度，尤其是线性核函数，很适合处理维度成百上千的分类问题。

此外，支持向量机仅能处理二分类问题，对于多分类问题，一个可行的办法是构建几个不同的支持向量机用于分类。这一点与 Logistic 回归模型类似。在实际处理分类问题时，一定要根据样本数据的特点选择合适的模型。对于支持向量机来说，要多尝试一些参数组合，直到找到最优的参数组合为止。

第 15 章 实现更高效的流程控制和高级循环

到目前为止，我们已经学习了如何使用 R 实现多种数据分析方法。本章将讨论如何使 R 程序能够更高效地运行。本章的主题是流程控制和高级循环，它们的实现方法与其他编程语言有较大的不同，通过阅读本章，读者将对 R 语言的特性有更加深入的了解。

15.1 R 中的流程控制

R 中的流程控制语句主要为 if 语句与 switch 语句，其中，if 语句在分支较少时使用，switch 语句在分支较多时使用，它们与其他编程语句中的使用方法并无太大不同，本节将讨论这两种语句在 R 中的使用方法。

15.1.1 if 语句的多种实现方法

在编写程序时，我们往往会遇到流程控制问题，即希望只有某条件成立时才执行某些程序，否则不执行。if 语句就是一条常用的流程控制语句，在前几节中，我们已经见过了许多 if 语句的例子，在这里专门总结了它的特性。

```
> if(TRUE) print("true")
[1] "true"
> if(FALSE) print("true")
> if(2>1) print("true")
[1] "true"
```

if 语句接受一个逻辑值作为判断条件，它的书写格式为“if(逻辑值)+ 执行语句”。逻辑值只有两种取值，“TRUE”或“FALSE”。上述代码给出了三个有关 if 语句的例子，在第一个例子中直接指定逻辑值为 TRUE，并指定执行语句为 print("true")，此时 if 语句执行了给出的执行语句，输出了 true。第二个例子中将 if 语句接受的逻辑值改为 FALSE，此时 R 并未返回任何结果，显然，if 语句只有在接受的逻辑值为 TRUE 时才会执行给定的执行语句，否则将跳过该语句。

通常情况下，我们并不会真的在 if 后的圆括号内直接给出 TRUE 或 FALSE，如果提前知道了判断条件的真假，似乎也就没必要使用 if 语句控制流程。if 后的圆括号内可以接受任何能够返回一个逻辑值的表达式，上述代码中第三个例子中给出的表达式为 2>1，由于 2 确实大于 1，因此，此处 if 语句接受的逻辑值就是 TRUE，R 执行了接下来的执行语句，返回了 true。

```
> if(FALSE) print("true") else print("false")
[1] "false"
```

if 语句能够单独拿来使用，也能够和 else 语句配套使用。上述代码在 if 语句后缀连了一个 else 语句，此时由于 if 语句接受的逻辑值为假，因此并不执行 if 语句对应的执行语句，转而去执行 else 语句对应的执行语句，即在屏幕上打印出 "false"。

else 语句并不需要判断条件，如果给 else 也用圆括号给定判断条件，R 将报错。if 语句和 else 语句显然满足互斥关系，在一组 ifelse 语句中，不可能既执行了 if 语句对应的执行语句，又执行了 else 语句对应的执行语句。if 语句和 else 语句也满足互补关系，if 语句对应的执行语句和 else 语句对应的执行语句必然有一条会被执行，不可能两者都不执行。

```
> if(FALSE) print("true")
> else print("false")
错误: 意外的'else' in "else"
```

由于 if 语句能够单独拿来使用，因此 if 语句和 else 语句必须放在同一行中才能被 R 识别为同一组语句。上述代码在第 1 行命令中输入了一句 if 语句，由于该语句接受的逻辑值为假，故 R 并未返回任何结果。在第 2 行命令中，R 认为这是一个新的命令，但 R 找不到与 else 对应的 if，故此处报错。

```
> if.else <- function(x){
+ if(x>1) print("x>1")
+ else print("x<=1")
+ }
> if.else(x=0)
[1] "x<=1"
```

除将 if 语句和 else 语句写入一行命令外，另一种解决办法是将 if 语句和 else 语句写入函数中。上述代码构建了一个函数 if.else，它接受一个参数 x ，当 x 大于 1 时返回 $x>1$ ，当参数 x 小于等于 1 时返回 $x<=1$ 。函数中的 if 语句和 else 语句分别写为两行命令。

调用 if.else 函数，并指定参数 x 为 0，此时判断语句 $x>1$ 的返回值为 FALSE，故函数执行了 else 语句对应的执行语句，返回了 "x<=1"，显然，此时拆分为两行的 if 语句和 else 语句工作良好。

15.1.2 ifelse 语句与花括号的结合

另一种经常与 if 语句搭配使用的符号是花括号。花括号能将多个语句组合为一个语句块，并告诉 R 将它们看为一个整体。

```
> if(TRUE){
+ print("true")
+ print("true true")
+ }
```

```
[1] "true"
[1] "true true"
```

上述代码在 if 语句后使用了花括号 {，在两行命令后输入了另一个花括号 }，R 会将花括号 {} 括起来的部分看作一个整体，上述代码中两个 print() 语句都返回了结果。显然，花括号 {} 有助于使 if 语句执行复杂的命令。

```
> if(FALSE){
+ print("true")
+ }else{
+ print("false")
+ }
[1] "false"
```

花括号 {} 同样能与 else 语句组合使用。上述代码在 if 语句对应的 } 符号后接上了 else 语句，R 将这 5 行命令视为一个整体，并返回了 false。} 符号与 else 语句不能拆成两行，否则 R 会认为 if 语句已经结束，并为 else 语句报错。如果不想把 } 符号与 else 语句连起来，则需要将它们写到一个函数中，并在需要控制流程时调用该函数。

```
> if(TRUE){
+ if(FALSE){
+ print("false")
+ }else{
+ print("true")
+ }
+ }
[1] "true"
```

花括号也允许嵌套调用 if 语句，上述代码首先给出了一个 if 语句，它接受逻辑值 TRUE，因此进入该分支，在该分支下，又有一个 if 语句，它接受的逻辑值为 FALSE，因此跳过该分支，进入下边的 else 分支，并输出 "true"，else 分支结束后，第一个 if 分支也将结束，R 返回流程控制语句的结果。

```
> if(1>2){
+ print("1>2")
+ }else if(1==2){
+ print("1=2")
+ }else{
+ print("1<2")
+ }
[1] "1<2"
```

除嵌套调用外，另一种有趣的用法是将几个 if 语句和 else 语句连接起来。上述代码中第一个 if 语句判断了 1>2 的真假，由于结果为假，故 R 跳过 if 语句对应的执行语句，进入第 3 行的 else 语句。第 3 行的 else 语句缀连着另一个 if 语句，它的判断条件为 1==2，即 1 与 2 相等，此时返回的逻辑值仍为假，故 R 再次跳过这个 if 语句对应的控制语句，进入最后一个 else 分支，并执行它对应的执行语句，返回 "1<2"。

```
> ifelse(1>2, "1>2", "1<=2")  
[1] "1<=2"
```

一种 R 特有的流程控制语句为 `ifelse` 语句。与上一个例子不同，这里的 `ifelse` 之间并无空格，而上一个例子中的 `else if` 则为两个单词。`ifelse` 语句接受三个参数，第一个参数为判断语句，用于返回逻辑值，当逻辑值为真时，返回第二个参数给出的语句，当逻辑值为假时，返回第三个参数给出的语句。

上述代码执行了 `ifelse` 函数，它的判断条件为 `1>2`，由于这个判断条件的结果为假，故 R 返回了第三个参数给出的结果。

```
> x <- c(1:5)  
> ifelse(x>2, "true", "false")  
[1] "false" "false" "true"  "true"  "true"
```

`ifelse` 语句的特殊之处在于它是一种向量化的函数，即它能对一个向量实现流程控制。上述代码首先创建了一个从 1 到 5 的序列 `x`，然后使用 `ifelse` 语句判断 `x` 是否大于 2，并在 `x` 大于 2 时返回 `true`，否则返回 `false`。R 返回了一个包含 5 个元素的向量，显然，`ifelse` 分别判断了 `x` 中每个元素与 2 的关系，并逐一返回了 `true` 或 `false`。

拆开的 `if` 语句与 `else` 语句并不接受长度多于 1 的逻辑值，因此对向量进行流程控制时 `ifelse` 是一种更合适的选择。

15.1.3 适合多分支情况的 `switch` 语句

`switch` 语句是 R 中最后一种常用的流程控制语句，当流程分支特别多时，`ifelse` 语句组合将非常复杂难懂，此时 `switch` 语句是更好的选择。

```
> switch(  
+ "a",  
+ a=1,  
+ b=2  
+ )  
[1] 1
```

`switch` 语句的书写格式较为简洁，对有 n 个分支的情况，`switch` 语句接受 $n+1$ 个参数。第一个参数为该次流程控制中应进入的分支，其余的 n 个参数为 n 个分支对应的执行语句。`switch` 语句的执行语句书写格式与 `ifelse` 语句相同，同样支持各种类型的执行语句，当 `switch` 语句中某一分支的执行语句多于一句时，花括号能帮助 R 将多个执行语句视为一个整体。

上述代码中 `switch()` 函数的第一个参数为 `a`，第二、三个参数分别给出了两个分支，这两个分支的匹配名称分别为 `a` 和 `b`，执行语句分别为 1 和 2。第一个参数与第一个分支的匹配名称完全一致，故 `switch` 语句跳入了第一个分支，并返回了结果 1。

```
> switch(  
+ "c",  
+ a=1,
```

```
+ b=2
+ )
```

switch 语句对第一个参数匹配不了任何分支的情况不做任何处理。上述代码中 switch() 函数的第一个参数为 c，它与两个分支的匹配名称都无法匹配，因此 R 并没有返回任何结果。

```
> switch(
+ "c",
+ a=1,
+ b=2,
+ 3
+ )
[1] 3
```

由于 switch 语句处理分支特别多的情况，有时难以人为确定第一个参数是否能匹配到某一分支中，为了 switch 语句返回 NULL，可以设置一个没有匹配名称的分支，令第一个参数在匹配不到其他分支时自动匹配入没有匹配名称的分支中。

上述代码中指定 switch() 函数的第一个参数为 c，并给出了三个分支，前两个分支的匹配名称分别为 a 和 b，最后一个分支则没有匹配名称，R 自动将 c 匹配入最后一个分支中，返回了 3。

```
> switch(
+ 1,
+ a=1,
+ b=2
+ )
[1] 1
> switch(
+ 4,
+ a=1,
+ b=2,
+ 3
+ )
```

在前面的几个例子中，switch 函数的第一个参数都加上了双引号，这表示第一个参数是字符形式，除使用字符形式的参数去逐一匹配每一分支的匹配名称外，也可以使用每一分支的序号作为匹配对象。

上述代码给出了两个例子，在第一个例子中，switch 语句的第一个参数为 1，R 将它匹配入了第一个分支中，返回了结果 1。在第二个例子中，switch 语句的第一个参数为 4，由于 switch() 函数只有三个分支，因此 R 并未返回任何结果。此时指定没有匹配名称的分支将不起作用，基本某一分支没有匹配名称，它也有一个自己的序号。

```
> switch(
+ "3",
+ "3"=1,
```

```
+ "4"=2
+ )
[1] 1
> switch(
+ 3,
+ "3"=1,
+ "4"=2
+ )
```

考虑当匹配名称为数字时，我们需要为匹配名称加上双引号，表明此处它们为字符。上述两个例子中的 `switch()` 函数都有两个分支，它们的匹配名称分别为 3 和 4，分支中的执行语句分别为 1 和 2。第一个例子中将第一个参数指定为带引号的 3，此时函数匹配到了匹配名称为 3 的分支，即第一个分支，并返回了结果 1。第二个例子中第一个参数为不带引号的 3，此时函数匹配到了第三个分支，由于函数仅有两个分支，故没有任何返回结果。

15.2 R 中的 for 循环、while 循环和 repeat 循环

本节将讨论三种常用的循环语句，它们不仅在 R 语言中能够使用，在其他编程语言中也是通用的。本节的内容同样是下一节的铺垫。

15.2.1 R 中的 for 循环和 while 循环

`for` 循环语句同样是在前文中出现了好多次的一种语句。`for` 循环语句的适用情况为循环次数已知的情况。它是最好用的循环语句。

```
> for(i in 1:5) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

上述代码给定的循环条件为“`i in 1:5`”，即 `for` 循环一共循环 5 次，每次循环中 i 的值分别为 1~5。循环语句的循环体为 `print(i)`，即将 i 的值输出到屏幕上。观察 R 的返回结果，R 一共返回了 5 个结果，`for` 循环每循环一次，R 就返回一个值，这 5 个数分别是 5 次循环中 i 的值。

```
> for(i in 1:5){
+ j <- 2*i
+ print(j)
+ }
[1] 2
[1] 4
[1] 6
```



```
[1] 8
[1] 10
```

当 for 循环语句的循环体多于一句时，同样可以使用花括号将它们括起来。上述代码将 for 循环的循环语句扩展为两句，第一句命令变量 j 为 i 的倍数，第二句命令输出了 j 的值。该循环同样循环了 5 次，每循环一次 R 将返回一个值，在这 5 次循环中， i 的值分别为 1、2、3、4、5，故 j 的值分别为 2、4、6、8、10。

```
> i <- c("a", "b", "c", "d")
> for(j in i) print(j)
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

for 循环语句的灵活之处在于它可以接受多种类型的循环条件，上述代码将一个字符向量 i 作为输入条件， i 中有 4 个元素 a、b、c、d，因此 for 循环将循环 4 次，每次循环中 j 的值都将迭代为 i 中对应位置的元素，如第一次循环中， j 的值为 i 中的第一个元素；第二次循环中， j 的值则为 i 中的第二个元素。for 循环的循环体为输出 j ，观察 R 的返回结果，该循环确实循环了 4 次，4 次循环中 j 的值分别为 a、b、c、d。

```
> n <- 0
> while(n<5){
+ print(n)
+ n <- n+1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
```

当循环次数不确定时，while 循环语句能够使用逻辑值确定是否执行循环或结束循环。while 循环语句接受一个逻辑值作为循环条件，当逻辑值为真时，执行循环体；当逻辑值为假时，结束循环。上述代码首先创建了一个变量 n ，并令它的值为 0，while 循环语句比较了 n 与 5 的大小，当 n 小于 5 时，执行循环体。

上述代码中的循环体包含如下两行命令：第 1 行命令输出了 n 的值，第 2 行命令令 n 增加 1。观察 R 的返回结果，while 循环语句一共循环了 5 次，当第五次循环执行完毕后， n 的值为 5，此时 $n<5$ 不再为真，因此循环结束。显然，使用 while 循环语句时要格外小心，如果循环条件是一个恒为真的值，循环体就会一直循环下去，除非中断当前计算。

```
> for(i in 1:5){
+ j <- i
+ while(j<3){
+ print(j)
+ j <- j+1
```

```
+ }
+ }
[1] 1
[1] 2
[1] 2
```

for 循环语句和 while 循环语句能够嵌套使用。上述代码中指定 for 循环的循环条件为“i in 1:5”，其循环体首先指定变量 j 的值为 i ，然后执行了 while 循环语句，while 循环语句的循环条件为 $j < 3$ ，循环体为输出 j ，并令 j 的值增加 1。

观察 R 的返回结果，在 for 循环的第一次循环中， i 为 1， j 也为 1，由于 $1 < 3$ ，故 while 循环执行了循环体，输出了 1，并令 j 的值为 2，2 仍然小于 3，故 while 循环再次执行循环体，输出 2，并将 j 的值增加为 3，此时 while 循环中止，for 循环进入第二次循环， i 的值更新为 2， j 的值也更新为 2，再次进入 while 循环，输出 2，并将 j 的值迭代为 3。for 循环的第三、四、五次循环都不再执行 while 循环体，也不再输出结果。

15.2.2 R 中的 repeat 循环

repeat 循环是一种与 while 循环非常类似的循环语句，但到目前为止，我们一次都未使用过 repeat 循环，这是因为 repeat 循环是一种非常容易出错的循环语句。

```
> repeat print("a")
```

repeat 循环语句不需要接受循环条件，它的循环条件放在循环体中，即至少循环一次后 repeat 循环才会跳出循环体。上述代码是一个 repeat 循环的例子，print(a) 表示输出 a，当然，这个语句并没有错，但它没有终止条件，因此会陷入死循环。中断当前计算后，会刷出一屏幕的 a。

```
> n <- 0
> repeat{
+ print(n)
+ n <- n+1
+ if(n>5) break
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

由于 repeat 循环语句需要在循环体中写入终止条件，因此 repeat 循环显然需要花括号将循环体括起来。上述代码首先创建了一个变量 n ，并令 n 的值为 0。repeat 循环语句中包含三行命令，第 1 行命令输出了 n ，第 2 行命令令 n 增加了 1，第 3 行命令则使用 if 语句判断 n 是否大于 5，当 n 大于 5 时，执行 break，即终止当前的 repeat 循环。

观察 R 的返回结果，由于输出 n 在将 n 增加 1 之前，故 R 从 0 开始返回，repeat 循

环一共循环了 6 次，在最后一次循环中，执行第 1 行命令时， n 的值为 5，执行第 2 行命令后， n 的值变为 6，满足 if 语句的条件，终止循环。

```
> n <- 0
> repeat{
+ n <- n+1
+ if(n==3) next
+ print(n)
+ if(n>5) break
+ }
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
```

repeat 循环的另一个关键词是 next，表示终止当前循环并进入下一次循环。在执行完上一个例子后， n 的值已迭代为 6，上述代码首先重新令 n 的值为 0，并再次进入 repeat 循环，其中循环体有 4 行命令，第 1 行命令将 n 的值增加 1；第 2 行命令表示当 n 的值为 3 时，中断当前循环；第 3 行命令输出了 n ；第 4 行命令则表示当 n 的值大于 5 时终止循环。

R 一共返回了 5 个值，但 repeat 循环语句仍循环了 6 次，当循环到第三次时， n 的值为 3，因此循环体执行到第 2 行命令就终止了第三次循环，并未向下执行 print() 函数，也未输出值。

如下两个例子并不能达到在输出结果中去掉一个值的目的。

```
> n <- 0
> repeat{
+ print(n)
+ n <- n+1
+ if(n==3) next
+ if(n>5) break
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

上述代码同样执行了 6 次循环，在执行第三次循环时， n 的值确实为 3，在执行循环体中第 3 行命令时也确实终止了当前循环，但在终止当前循环时已经执行了 print() 函数，因此并不能达到在输出结果中去掉一个值的目的。

```
> n <- 0
> repeat{
```

```
+ if(n==3) next
+ print(n)
+ n <- n+1
+ if(n>5) break
+ }
[1] 0
[1] 1
[1] 2
```

上述代码将陷入死循环。在前 3 次循环时，repeat 循环语句执行了全部循环体，输出了 0、1 和 2，并将 n 迭代到了 3。在第 4 次循环时，repeat 循环执行了第 1 行命令便终止了当前循环，跳入第 5 次循环，此时并未给 n 的值增加 1，因此在第 5 次循环时 n 的值仍为 3，repeat 循环执行了第 1 行命令后再次终止当前循环，跳入第六次循环，如此便陷入了死循环。

15.3 apply 家族中的循环函数

有关循环语句，R 提供了以 apply() 函数为首的 apply 家族。本节将讨论 apply 家族中主要成员的用法，并指出 apply 家族的优越之处，在 R 中，我们应当尽量使用 apply 家族中的循环函数实现循环功能，以保证 R 的性能。

15.3.1 R 中的 apply() 函数

for 循环语句和 while 循环语句在其他编程语言中是常用的语句，但在 R 中，我们更倾向于使用 apply 家族中的函数实现循环。这是由于 R 是一种向量化的语言，使用向量化循环方式处理数据所需要的时间比使用 for 循环语句要少一到两个数量级。考虑到 R 的运行速度本来就很慢，因此抛弃会使它的速度更慢的 for 循环是非常合理的选择。

```
> x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
> dimnames(x)[[1]] <- letters[1:8]
> x
  x1 x2
a  3  4
b  3  3
c  3  2
d  3  1
e  3  2
f  3  3
g  3  4
h  3  5
> apply(x, 2, mean)
x1 x2
 3  3
```

可以这样来理解向量化的循环语句，所谓向量化，就是对一个向量中的元素逐一应用某种函数。15.1 节有关 `ifelse()` 函数的介绍中已经展示了如何对一个向量中的元素逐一实现流程控制功能。而 `apply` 家族所能实现的就是对一个向量中的元素逐一实现循环功能。

上述代码首先创建了一个矩阵，`cbind()` 函数将两个向量按列联合，`dimnames()` 函数则表示给矩阵 x 按行命名，`letters` 中按顺序存储了 26 个小写字母，故 x 的 8 个行分别得到名称 `a`、`b`、`c`、`d`、`e`、`f`、`g`、`h`。上述代码中第 4 行命令调用了 `apply()` 函数，其中，第一个参数表示 x 为被应用对象；第二个参数取值为 2，表示对 x 的第二维进行循环，即按列循环，它取 1 时表示按行循环；第三个参数则是被循环函数。 x 有两列，故 `apply()` 函数执行了两次循环，`R` 返回了矩阵 x 的两个列均值。

```
> apply(x, 2, sort)
      x1 x2
[1,]  3  1
[2,]  3  2
[3,]  3  2
[4,]  3  3
[5,]  3  3
[6,]  3  4
[7,]  3  4
[8,]  3  5
```

上述代码修改了在 x 上应用的函数，`sort()` 函数表示排序，显然，`R` 将矩阵 x 中的两列数值进行了排序，并返回了排序后的新矩阵。

```
> col.sums <- apply(x, 2, sum)
> row.sums <- apply(x, 1, sum)
> rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))
      x1 x2 Rtot
a      3  4    7
b      3  3    6
c      3  2    5
d      3  1    4
e      3  2    5
f      3  3    6
g      3  4    7
h      3  5    8
Ctot 24 24   48
```

`col.sums` 为 `apply()` 函数对 x 中元素按列求和的结果，`row.sums` 为 `apply()` 函数对 x 中元素按行求和的结果，`cbind` 将 `row.sums` 接到了 x 的右边，`rbind` 将 `col.sums` 和 `sum(col.sums)` 接到了 x 的下边。观察 `R` 的返回结果，此时 `R` 返回了一个 9 行 3 列的矩阵，列 `Rtot` 的前 8 个元素是矩阵 x 的每一行的和，行 `Ctot` 的前两个元素则是矩阵 x 的每一列的和，列 `Rtot` 与行 `Ctot` 相交叉的元素则是矩阵 x 中全体元素的和。

```

> z <- array(1:24, dim = 2:4)
> z
, , 1

      [,1] [,2] [,3]
[1,]     1     3     5
[2,]     2     4     6

, , 2

      [,1] [,2] [,3]
[1,]     7     9    11
[2,]     8    10    12

, , 3

      [,1] [,2] [,3]
[1,]    13    15    17
[2,]    14    16    18

, , 4

      [,1] [,2] [,3]
[1,]    19    21    23
[2,]    20    22    24

> apply(z, 3, function(x) seq_len(max(x)))
[[1]]
[1] 1 2 3 4 5 6

[[2]]
[1] 1 2 3 4 5 6 7 8 9 10 11 12

[[3]]
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

[[4]]
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

```

最后，来看一个复杂一些的例子，`apply()` 函数能接受一个 R 提供的函数作为循环函数，分析人员也能为它指定一个人为创建的新函数。

上述代码首先创建了一个包含 1~24 的序列的数组 `z`，它有三个维度，这三个维度分别包含 2、3、4 个元素，当 `z` 按照第三维分组时，我们能得到 4 个元素组，分别为数列 1,2,3,4,5,6、数列 7,8,9,10,11,12，数列 13,14,15,16,17,18，以及数列 19,20,21,22,23,24。

上述代码的最后一行命令调用了 `apply()` 函数，它有三个参数，前两个参数表示按照 `z` 的第三维进行分组，并分别对这 4 组元素应用循环函数。`apply()` 函数的第三个参数给

出了循环函数，在这里我们创建了一个临时函数 `function()`，它接受一个参数 x 的传入，并只有一句循环体。

由于 z 的第三维有 4 个组，因此循环将进行 4 次，每次向临时函数 `function()` 中传入的 x 分别为这 4 个组中的全体元素。`max()` 函数求出了 x 的最大值，在这 4 次循环中，最大值分别为 6、12、18 和 24，`seq_len()` 函数则创建了从 1 到 `max(x)` 的序列，R 返回了最终生成的 4 个序列。

15.3.2 R 中的 `lapply()` 函数和 `sapply()` 函数

`apply()` 函数提供了一种非常好用的向量化循环语句，在 `apply()` 函数的基础上，又引申出 `lapply()` 函数和 `sapply()` 函数，其中，`lapply()` 函数在对列表元素进行循环时特别有用。

```
> x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
> x
$a
[1] 1 2 3 4 5 6 7 8 9 10

$beta
[1] 0.04978707 0.13533528 0.36787944 1.00000000 2.71828183
7.38905610 20.08553692

$logic
[1] TRUE FALSE FALSE TRUE

> lapply(x, mean)
$a
[1] 5.5

$beta
[1] 4.535125

$logic
[1] 0.5
```

上述代码首先使用 `list()` 函数创建了一个列表 x ，它包含三个元素：`a`、`beta` 和 `logic`，第 2 行命令查看了 x 中的数据结构。第 3 行命令调用了 `lapply()` 函数，第一个参数指定 x 为应用对象，第二个参数指定 `mean()` 函数为应用函数。

列表 x 中共有三个元素，`lapply()` 函数对这三个元素分别应用了 `mean` 函数，即求出了 x 中每个元素的均值， x 中 `a` 和 `beta` 均为数值型元素，但 `logic` 为逻辑型元素，`mean` 函数将 `logic` 中的 `TRUE` 视为 1，`FALSE` 视为 0，计算了均值。

```
> lapply(x, quantile, probs = 1:3/4)
$a
25% 50% 75%
```

```

3.25 5.50 7.75

$beta
      25%      50%      75%
0.2516074 1.0000000 5.0536690

$logic
      25% 50% 75%
0.0 0.5 1.0

```

`lapply()` 函数的灵活之处还在于它可以根据被应用函数调整参数。上述代码中，`lapply()` 函数指定了三个参数，其中，前两个参数指定 x 为应用对象；`quantile()` 函数为应用函数，它起到求分位数的作用；参数 `probs` 存在于 `quantile()` 函数中，用于指定被计算的分位数，由于运算符 “:” 的优先级大于 “/”，因此此处 `probs` 为 1/4、2/4、3/4。

观察 R 的返回结果，`lapply()` 函数给出了 x 中 `a`、`beta`、`logic` 的下四分位数、中位数和上四分位数，在这里 `logic` 元素中的 `TRUE` 和 `FALSE` 同样被视为 1 和 0 处理。

```

> sapply(x, quantile)
      a      beta logic
0%    1.00  0.04978707  0.0
25%    3.25  0.25160736  0.0
50%    5.50  1.00000000  0.5
75%    7.75  5.05366896  1.0
100% 10.00 20.08553692  1.0

```

`sapply()` 函数的用法与 `lapply()` 函数非常相似，上述代码调用了 `sapply()` 函数，对 x 中的元素逐个应用了 `quantile()` 函数，由于并未在 `sapply()` 函数中进一步指定 `probs` 参数，此处返回了默认的五分位数。与 `lapply()` 函数不同的是，`sapply()` 函数默认返回一个向量或矩阵，而 `lapply()` 函数返回的则是列表。在 `sapply()` 函数中，只有当返回结果的长度不一，不能生成矩阵时，才会生成一个列表。

```

> v <- structure(10*(5:8), names = LETTERS[1:4])
> v
  A  B  C  D
50 60 70 80
> f2 <- function(x, y) {outer(rep(x, length.out = 3), y)}
> a2 <- sapply(v, f2, y = 2*(1:5), simplify = "array")
, , A

      [,1] [,2] [,3] [,4] [,5]
[1,]  100  200  300  400  500
[2,]  100  200  300  400  500
[3,]  100  200  300  400  500

, , B

      [,1] [,2] [,3] [,4] [,5]

```



```
[1,] 120 240 360 480 600
[2,] 120 240 360 480 600
[3,] 120 240 360 480 600
```

```
, , C
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,] 140  280  420  560  700
[2,] 140  280  420  560  700
[3,] 140  280  420  560  700
```

```
, , D
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,] 160  320  480  640  800
[2,] 160  320  480  640  800
[3,] 160  320  480  640  800
```

最后，同样来看一个较为复杂的例子。上述代码首先使用 `structure()` 函数构建了一个数值型向量 `v`，其值为 `10*(5:8)`，即 10 与 5、6、7、8 分别相乘，`names` 参数指定 4 个值的名字为 `LETTERS` 中的前 4 个元素，即 A、B、C、D，第 2 行命令查看了 `v` 中存储的内容。

第 3 行命令创建了一个函数 `f2`，它接受两个形参 `x`、`y`，函数体为 `outer(rep(x, length.out = 3), y)`，此处使用花括号将函数体括起，当函数体中的语句多于二句时，花括号是必需的。`outer()` 函数用于计算外积，`rep(x, length.out = 3)` 和 `y` 是被计算的对象，其中，`rep(x, length.out = 3)` 表示将 `x` 重复 3 次，即生成数列 `x,x,x`。

最后一行命令调用了 `sapply()` 函数，它对 `v` 中的每一元素应用了 `f2` 函数，由于 `v` 中有 4 个元素：50、60、70 和 80，因此 `f2` 函数循环执行了 4 次。

在第一次循环中，`f2` 接受 50 作为 `x` 的值，`sapply()` 函数指定 `y` 为 `2*(1:5)`，即数列 2,4,6,8,10，`rep(x, length.out = 3)` 在第 1 次循环中生成了序列 50,50,50，`outer` 函数则计算了数列 50,50,50 和数列 2,4,6,8,10 的外积，即 R 返回结果中元素 A 中的值。

在第 2、3、4 次循环中，`y` 的值并未发生变化，`x` 的值分别变为了 60、70 和 80，`rep(x, length.out = 3)` 也分别变为了数列 60,60,60、数列 70,70,70，以及数列 80,80,80。上述代码同时指定 `simplify = "array"`，即返回一个数组，由于元素 A、B、C、D 的宽度相同，因此去掉这个参数后，`sapply` 将返回一个矩阵。

15.3.3 R 中的 `tapply()` 函数

到目前为止，`apply()` 函数、`lapply()` 函数和 `sapply()` 函数提供的分组循环的方式都非常简单，即按照被应用对象的维数进行分组循环。`tapply()` 函数提供了更灵活、更好用的循环方式，当然，也更难以理解、更容易出错。

```
> groups <- as.factor(c(13,13,9,11,9))
> groups
```

```
[1] 13 13 9 11 9
Levels: 9 11 13
> tapply(groups, groups, length)
 9 11 13
 2  1  2
```

上述代码首先创建了一个序列 13,13,9,11,9，并将它转换为了因子类型。查看 `groups`，它显然有三个因子类别，分别为 9、11 和 13。第 3 行命令调用了 `tapply` 函数，它指定了三个参数，第一个 `groups` 指定 `groups` 为被循环对象，第二个 `groups` 为指针参数，第三个 `length` 指定被循环函数为 `length()` 函数，即计算长度。

由于 `groups` 中有三个因子等级，因此 `tapply()` 函数将按照这三个因子等级分别循环三次。在第一次循环中，`tapply()` 函数按照第一个等级 9 取出了 `groups` 中值为 9 的全部元素，由于 `groups` 有两个值为 9 的元素，故 `length` 函数将返回结果 2；第二次循环中 `tapply()` 函数取出了 `groups` 中值为 11 的全部元素，`length` 的返回结果为 1；同理，`length` 的第三个返回结果为 2。

```
> head(warpbreaks)
  breaks wool tension
1     26    A       L
2     30    A       L
3     54    A       L
4     25    A       L
5     70    A       L
6     52    A       L
> summary(warpbreaks[,c(2:3)])
 wool    tension
A:27    L:18
B:27    M:18
        H:18
> tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
      tension
 wool    L    M    H
  A 401 216 221
  B 254 259 169
```

上述代码中 `head()` 函数查看了数据集 `warpbreaks` 的前 6 行数据，它存放了三列变量，第 1 列是数值型变量，第 2、3 列是分类变量。`summary()` 函数查看了 `warpbreaks` 中第 2、3 行变量的摘要，变量 `wool` 有两个分类，变量 `tension` 有三个分类。

第 3 行命令中 `tapply()` 函数将 `warpbreaks` 中的 `breaks` 变量作为应用对象；将 `warpbreaks` 中除去第 1 列的其他变量作为指针参数，即第 2、3 列变量；第三个参数表明被应用的函数为 `sum`，即求和。

在这里，由于第 2 列变量有两个变量，第 3 列变量有三个变量，此二者交叉后即得到 6 个指针，分别是 AL、AH、AM、BL、BM 和 BH，故 `tapply` 将执行 6 次循环，第一次循环中，`tapply` 筛选出 `breaks` 对应的 `wool` 为 A、`tension` 为 L 的全体数据，并对其求

和;第二次循环则筛选出 `wool` 为 A、`tension` 为 H 的全部 `breaks`, 并对其求和, 以此类推, 最终得到了 2 行 3 列的返回结果。

```
> tapply(warpbreaks$breaks, warpbreaks[, 3, drop = FALSE], sum)
tension
  L    M    H
655 475 390
```

上述代码在上一个例子的基础上将指针参数由 `tool` 和 `tension` 修改为 `tension`, 减去一个变量后, 此时指针参数仅包含三个值, 因此 `tapply()` 函数仅执行了三次循环, 计算出类别 L、M、H 分别对应的 `breaks` 的和。在指针参数中指定 `drop` 为 F, 即不要降维, 如果去掉这个函数, 由于最终结果只有一行数据, R 将自动返回一个向量, 返回结果中的标签 “`tension`” 将不被显示。

```
> ind <- list(c(1, 2, 2), c("A", "A", "B"))
> ind
[[1]]
[1] 1 2 2

[[2]]
[1] "A" "A" "B"
> table(ind)
      ind.2
ind.1 A B
  1  1 0
  2  1 1
> tapply(1:3, ind)
[1] 1 2 4
> tapply(1:3, ind, sum)
  A  B
1 1 NA
2 2  3
```

有关 `tapply()` 函数的最后一个例子更加明白地揭示了 `tapply()` 函数的工作原理。上述代码首先创建了一个列表 `ind`, 它的第一个元素为 1,2,2, 第二个元素为 "A", "A", "B", `table` 函数查看了 `ind` 的列联表。当 `ind` 作为指针参数时, `ind` 能组合出 4 种指针元素, 在统计指针元素时, `tapply()` 函数先循环指针参数的第一维, 后循环第二维, 故 `ind` 的 4 个指针元素分别是 1A、2A、1B 和 2B, `tapply()` 函数将按照这个顺序进行循环。

第 4 行命令 `tapply(1:3, ind)` 并未指定被应用函数, 因此它返回了一个索引向量, 在第一次循环中 `tapply()` 函数搜索了 `ind` 中的第一组元素 1A (这里指 `ind` 中位于第一个位置的元素, 而不是第一个指针元素), 并返回了它对应的指针元素的顺序数, 即 1; 在第二次循环中, `tapply()` 函数搜索了 `ind` 中的第二组元素 2A。显然, `tapply()` 函数的第一个参数和第二个参数需要保持同一长度。

第 4 行命令 `tapply(1:3, ind, sum)` 在上一句代码的基础上添加了被应用函数为 `sum`, 即求和, 此时 `tapply` 循环了 4 次, 在第一次循环中, `tapply` 查看了 `ind` 中值为 1A 的元素,

ind 中只有第一组元素为 1A，故 `tapply` 抽取了序列 1,2,3 的第一个元素，并返回了 1；第三次循环中由于 ind 中没有值为 1B 的元素，故值为 NA；第四次循环中与 2B 相同元素位于第三个位置，故返回了 1,2,3 中的第三个元素，即 3。

15.3.4 R 中的 `mapply()` 函数

从某种意义上来说，`mapply()` 函数与 `tapply()` 函数非常相似，它们都借助指针参数给定循环条件，这是一种灵活到不好掌握的方式。与其他 `apply` 家族的函数相比，`mapply()` 函数的另一个特点是它的参数顺序与其他函数恰好相反。

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

上述代码调用了 `mapply()` 函数，它有三个参数，第一个参数指定被应用的函数为 `rep`，即重复；第二个参数指定被应用的对象为序列 1,2,3,4；第三个参数为 `rep` 的补充，指定重复的次数为 4,3,2,1。观察 R 的返回结果，`mapply()` 函数一共执行了 4 次循环，分别将 1,2,3,4 重复了 4,3,2,1 次，并生成了序列。

```
> mapply(rep, times = 1:4, x = 4:1)
[[1]]
[1] 4

[[2]]
[1] 3 3

[[3]]
[1] 2 2 2

[[4]]
[1] 1 1 1 1
```

上述代码指定被应用的函数为 `rep`，即重复，第二个参数指定重复的次数为序列 1,2,3,4，第三个参数指定被应用的对象为 4,3,2,1。观察 R 的返回结果，它一共执行了 4 次循环，将 4,3,2,1 分别重复了 1,2,3,4 次。与上一个例子相比，本例写明了后两个参数的参数名，破坏了默认的参数顺序。

```
> mapply(rep, times = 1:4, MoreArgs = list(x = 42))
[[1]]
[1] 42

[[2]]
[1] 42 42

[[3]]
[1] 42 42 42

[[4]]
[1] 42 42 42 42
```

上述代码在上一例的基础上将参数 x 修改为参数 `MoreArgs`，它用于指定最后一个参数，`times` 参数有 4 个值，R 自动循环了 4 次，在每次循环中被应用对象都是 42。参数 `MoreArgs` 总是用于指定最后一个参数，在不同的例子中，它能够代替不同的参数。

```
> mapply(function(x, y) seq_len(x) + y,
+ c(a = 1, b = 2, c = 3),
+ c(A = 10, B = 0, C = -10))
$a
[1] 11

$b
[1] 1 2

$c
[1] -9 -8 -7
```

上述代码在被应用函数的位置添加了一个临时函数，它有两个形参 x 、 y ，函数体为 `seq_len(x) + y`。`c(a = 1, b = 2, c = 3)` 给出了参数 x 的取值，`c(A = 10, B = 0, C = -10)` 给出了参数 y 的取值。它们的长度都为 3，显然，`mapply()` 函数将循环三次。

在第一次循环中， x 为 1， y 为 10，`seq_len(x)` 生成一个从 1 到 n 的序列，此时 n 为 1，故 `seq_len(x)` 为 1，而 `seq_len(x) + y` 则为 11；第二次循环中， x 为 2， y 为 0，`seq_len(x)` 为 1,2，而 `seq_len(x) + y` 则为 1,2；第三次循环中， x 为 3， y 为 -10，`seq_len(x)` 为 1,2,3，而 `seq_len(x) + y` 则为 -9，-8，-7。函数 `mapply` 自动将形参 x 的取值的名称 `a`、`b`、`c` 作为每一个元素的名称。

```
> word <- function(C, k) paste(rep.int(C, k), collapse = "")
> str(mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE))
List of 6
 $ A: chr "AAAAAA"
 $ B: chr "BBBBB"
 $ C: chr "CCCC"
 $ D: chr "DDD"
 $ E: chr "EE"
 $ F: chr "F"
```

上述代码创建了一个函数 `word`，它接受两个参数 `C`、`k`，`paste()` 函数为连接，`rep.int()` 将 `C` 重复 `k` 次，参数 `collapse` 指定连接这 `k` 个 `C` 的符号为 `""`，即空，`mapply` 的被应用函数为 `word`，被应用对象为 `LETTERS[1:6]`，即大写字母 A、B、C、D、E、F，它作为 `C` 传入函数 `word`，序列 6:1 作为 `k` 传入 `word`，`word` 一共执行 6 次，`SIMPLIFY` 为 F，返回的即为列表，`str()` 函数查看了 `mapply()` 函数返回结果的结构。

15.4 更多的高级循环函数

本节的内容是 `apply` 函数族的补充内容。本节将讨论三种其他的常用循环函数，它们同样是向量化的循环函数，在有些时候，调用这些函数完成循环更加方便。

15.4.1 R 中的 `replicate()` 函数和 `sweep()` 函数

早在我们学习如何创建序列时我们就已经接触了第一个循环函数：`rep()` 函数。当然，它只能实现非常简单的循环功能，即把一个数值循环几次。与它非常相似的还有 `replicate()` 函数，它能实现稍复杂的循环功能。

```
> rep(1,5)
[1] 1 1 1 1 1
> rep(c(1,2),5)
[1] 1 2 1 2 1 2 1 2 1 2
> replicate(5, mean(rexp(10)))
[1] 0.8265642 0.8086543 0.4252999 1.0924022 1.3071843
```

上述代码中第 1 行命令使用 `rep()` 函数将数字 1 重复了 5 次，第 2 行命令则把序列 1,2 重复了 5 次。第 3 行命令使用 `replicate()` 函数将 `mean(rexp(10))` 重复了 5 次，其中，`rexp(10)` 表示生成 10 个服从指数分布的随机数，`mean()` 函数则对其取得了均值。值得注意的是，`rep()` 函数与 `replicate()` 函数的默认参数顺序是相反的。

```
> head(attitude)
  rating complaints privileges learning raises critical advance
1     43         51         30      39      61      92       45
2     63         64         51      54      63      73       47
3     71         70         68      69      76      86       48
4     61         63         45      47      54      84       35
5     81         78         56      66      71      83       47
6     43         55         49      44      54      49       34
> med.att <- apply(attitude, 2, median)
> med.att
  rating complaints privileges learning raises critical advance
 65.5      65.0      51.5      56.5      63.5      77.5      41.0
> head(sweep(data.matrix(attitude), 2, med.att))
  rating complaints privileges learning raises critical advance
[1,] -22.5         -14      -21.5    -17.5     -2.5      14.5       4
```

[2,]	-2.5	-1	-0.5	-2.5	-0.5	-4.5	6
[3,]	5.5	5	16.5	12.5	12.5	8.5	7
[4,]	-4.5	-2	-6.5	-9.5	-9.5	6.5	-6
[5,]	15.5	13	4.5	9.5	7.5	5.5	6
[6,]	-22.5	-10	-2.5	-12.5	-9.5	-28.5	-7

另一种高级循环函数是 `sweep()` 函数，它只能实现相减的功能。上述代码中前两行命令使用 `apply()` 函数对 `attitude` 数据集按列求得均值，并赋给变量 `med.att`。

第 4 行命令调用了 `sweep()` 函数，它有三个参数，第一个参数指定被应用对象为数据框类型的 `attitude`，第二个参数指定对 `attitude` 的第二个维度应用函数，第三个参数表明 `med.att` 为减法函数的另一个应用对象。观察 R 的返回结果，`attitude` 的每一列均减去了该列的均值。

```
> A <- array(1:24, dim = 4:2)
> A
, , 1
      [,1] [,2] [,3]
[1,]     1     5     9
[2,]     2     6    10
[3,]     3     7    11
[4,]     4     8    12

, , 2
      [,1] [,2] [,3]
[1,]    13    17    21
[2,]    14    18    22
[3,]    15    19    23
[4,]    16    20    24
> sweep(A, 1, 5)
, , 1
      [,1] [,2] [,3]
[1,]    -4     0     4
[2,]    -3     1     5
[3,]    -2     2     6
[4,]    -1     3     7

, , 2
      [,1] [,2] [,3]
[1,]     8    12    16
[2,]     9    13    17
[3,]    10    14    18
[4,]    11    15    19
```

上述代码是另一个有关 `sweep()` 函数的例子，前两行命令创建了数组 `A`，并查看了 `A` 中的数据结构，第三行命令对 `A` 按行应用了 `sweep()` 函数，`sweep()` 函数的第三个参数为 5，`A` 中的每一个值都减去了 5。显然，`sweep()` 函数也接受长度为 1 的被减元素。

```
> A.min <- apply(A, 1, min)
> A.min
[1] 1 2 3 4
> sweep(A, 1, A.min)
, , 1
```

	[,1]	[,2]	[,3]
[1,]	0	4	8
[2,]	0	4	8
[3,]	0	4	8
[4,]	0	4	8

```
, , 2
```

	[,1]	[,2]	[,3]
[1,]	12	16	20
[2,]	12	16	20
[3,]	12	16	20
[4,]	12	16	20

上述代码中第 1 行命令对 `A` 中的第一个维度执行了 `min` 函数，即抽取最小值，`A` 中第一个维度共有 4 个元素，这 4 个元素分别是数列 1,5,9,13,17,21、数列 2,6,10,14,18,22、数列 3,7,11,15,19,23，以及数列 4,8,12,16,20,24，对它们取最小值后得到 1,2,3,4。

第 3 行命令对 `A` 按行减去了 `A.min`，第 1 行均减去 1，第 2 行均减去 2，第 3 行均减去 3，第 4 行均减去 4。

```
> sweep(A, 1:2, apply(A, 1:2, median))
, , 1
```

	[,1]	[,2]	[,3]
[1,]	-6	-6	-6
[2,]	-6	-6	-6
[3,]	-6	-6	-6
[4,]	-6	-6	-6

```
, , 2
```

	[,1]	[,2]	[,3]
[1,]	6	6	6
[2,]	6	6	6
[3,]	6	6	6
[4,]	6	6	6

上述代码中对 A 中 1:2 维减去了 `apply(A, 1:2, median)`，A 中按照 1:2 维抽取数据后得到 12 组元素，分别为 1,13、5,17、9,21、2,14、6,18、10,22、3,15、7,19、11,23、4,16、8,20、12,24。而 `apply(A, 1:2, median)` 分别对这 12 组数据取了均值，`sweep` 则令 A 中这 12 组数据减去了对应的均值。

15.4.2 R 中的 `aggregate()` 函数

`aggregate()` 函数是本章要介绍的最后一个高级循环函数，它的功能与 `apply` 函数族较为相似，但其用法更加丰富。

```
> head(state.x77)
      Population Income Illiteracy Life Exp Murder HS Grad Frost Area
Alabama      3615   3624         2.1   69.05   15.1   41.3    20 50708
Alaska       365   6315         1.5   69.31   11.3   66.7   152 566432
Arizona     2212   4530         1.8   70.55    7.8   58.1    15 113417
Arkansas     2110   3378         1.9   70.66   10.1   39.9    65 51945
California   21198  5114         1.1   71.71   10.3   62.6    20 156361
Colorado     2541  4884         0.7   72.06    6.8   63.9   166 103766

> head(state.region)
[1] South West  West  South West  West
Levels: Northeast South North Central West

> aggregate(state.x77, list(Region = state.region), mean)
      Region Population   Income Illiteracy Life Exp   Murder   HS Grad
Frost   Area
1 Northeast   5495.111 4570.222   1.000000 71.26444  4.722222 53.96667
132.7778 18141.00
2 South      4208.125 4011.938   1.737500 69.70625 10.581250 44.34375
64.6250 54605.12
3 North Central 4803.000 4611.083   0.700000 71.76667  5.275000 54.51667
138.8333 62652.00
4 West       2915.308 4702.615   1.023077 71.23462  7.215385 62.00000
102.1538 134463.00
```

`aggregate()` 函数同样需要一个被应用参数和一个指针参数，上述代码中第 3 行命令调用了 `aggregate()` 函数，它对 `state.x77` 按照 `state.region` 分组，并计算了每一组内的均值。`state.region` 中有 4 个元素，分别是 Northeast、South、North Central 和 Westaggregate。

`aggregate()` 函数一共执行了 4 次循环，在第一次循环中查找了 `state.region` 中的所有 Northeast 元素，并记录了它们的位置，然后抽取出位于相同位置的 `state.x77` 元素，并作为 `mean` 函数的作用对象。其他三次循环则对 South、North Central 和 Westaggregate 完成了相同的工作。

```
> aggregate(state.x77,
+           list(Region = state.region,
+               Cold = state.x77[, "Frost"] > 130),
+           mean)
```

	Region	Cold	Population	Income	Illiteracy	Life Exp	Murder
HS Grad	Frost	Area					
1	Northeast	FALSE	8802.8000	4780.400	1.1800000	71.12800	5.580000
52.06000	110.6000	21838.60					
2	South	FALSE	4208.1250	4011.938	1.7375000	69.70625	10.581250
44.34375	64.6250	54605.12					
3	North Central	FALSE	7233.8333	4633.333	0.7833333	70.95667	8.283333
53.36667	120.0000	56736.50					
4	West	FALSE	4582.5714	4550.143	1.2571429	71.70000	6.828571
60.11429	51.0000	91863.71					
5	Northeast	TRUE	1360.5000	4307.500	0.7750000	71.43500	3.650000
56.35000	160.5000	13519.00					
6	North Central	TRUE	2372.1667	4588.833	0.6166667	72.57667	2.266667
55.66667	157.6667	68567.50					
7	West	TRUE	970.1667	4880.500	0.7500000	70.69167	7.666667
64.20000	161.8333	184162.17					

`aggregate()` 函数同样接受多个指针参数。上述代码在上一个例子的基础上添加了分组条件 `state.x77[, "Frost"] > 130`，该条件是一个判断语句，返回逻辑值 T 或 F，两个条件组合后得到 8 个分组条件，因此，`aggregate()` 函数共执行 8 次循环，但它返回了一个 7*11 的矩阵，其中 South 条件下的 Frost 总是小于等于 130，故返回结果中没有 South、TRUE 的分组。

```
> aggregate(weight ~ feed, data = chickwts, mean)
      feed  weight
1  casein 323.5833
2 horsebean 160.2000
3  linseed 218.7500
4  meatmeal 276.9091
5  soybean 246.4286
6 sunflower 328.9167
```

上述代码使用公式的形式指定被应用对象和分组条件，公式书写格式为“被应用对象 `x~` 分组条件 `by`”，`weight` 和 `feed` 都是数据集 `chickwts` 中的变量，参数 `data` 指明 `chickwts` 为变量的来源数据集，`mean` 则指明被应用函数，`feed` 一共有 6 种取值，`aggregate()` 函数循环了 6 次，计算了每种取值下对应的 `weight` 的均值，这种方便的书写方式是 `aggregate()` 函数与 `apply` 函数族的主要不同，也是它受欢迎的主要原因之一。

```
> aggregate(cbind(ncases, ncontrols) ~ alcgp + tobgp, data = esoph, sum)
      alcgp  tobgp  ncases ncontrols
1 0-39g/day 0-9g/day      9        261
2   40-79 0-9g/day     34        179
3   80-119 0-9g/day     19         61
4    120+ 0-9g/day     16         24
5 0-39g/day 10-19      10         84
6   40-79 10-19      17         85
7   80-119 10-19      19         49
```

8	120+	10-19	12	18
9	0-39g/day	20-29	5	42
10	40-79	20-29	15	62
11	80-119	20-29	6	16
12	120+	20-29	7	12
13	0-39g/day	30+	5	28
14	40-79	30+	9	29
15	80-119	30+	7	12
16	120+	30+	10	13

`aggregate()` 函数的公式参数同样接受多个分组条件或多个被循环对象。上述代码中 `aggregate()` 函数有三个参数，第一个参数中的 `cbind(ncases, ncontrols)` 部分表明 `ncases` 和 `ncontrols` 均为函数的应用对象，`alcgp + tobgrp` 部分则表明 `alcgp` 和 `tobgrp` 都作为分组条件，参数 `data` 指定了数据来源，最后一个参数则表明被应用的函数为 `sum`，即求和。

观察 R 的返回结果，显然 `alcgp` 有 4 个分类，`tobgrp` 有 4 个分类，它们一共组合得到 16 种分组，因此，`aggregate()` 函数共循环了 16 次，分别计算了 `ncases` 和 `ncontrols` 中满足这 16 种分组条件的元素的和。此外，`aggregate()` 函数中的第一个参数写为公式时，也接受符号 “.” 表示数据来源中的全体变量，其他更多的使用方法与方差分析、回归分析中公式参数的使用方法保持一致。

第 16 章 R 代码的调试与优化

本章的主题是 R 代码的调试与优化，到现在为止，我们已经学习了许多种模型，这些模型已经能够解决绝大部分的实际问题。本章并未尝试继续介绍新模型，而是讨论如何优化代码。通过阅读本小节，读者将了解到如何写出正确、优美的代码。

16.1 R 代码的常见信息与警告

本节将讨论 R 中正常情况下的返回信息，并讨论第一种非正常的程序状态，即警告。本节还将讨论如何处理警告信息，包括忽略警告信息或提高对警告信息的警惕。

16.1.1 R 代码的正常信息与警告

通常来说，R 程序有三种状态，当处于正常状态时，R 程序能够正常运行，并给出正常回应。正常回应包括不给出回应或给出正确的回应。R 软件秉承“没有消息就是好消息”的设计原则，并不会特意在代码成功运转后给出提示信息。

```
> x <- c(1:10)
> head(x)
[1] 1 2 3 4 5 6
> x
[1] 1 2 3 4 5 6 7 8 9 10
> print(x)
[1] 1 2 3 4 5 6 7 8 9 10
> message(x)
12345678910
> (x <- c(1:10))
[1] 1 2 3 4 5 6 7 8 9 10
```

例如，上述代码首先成功创建了变量 x ，但 R 并不会专门提示我们变量创建成功，不过 R 提供了几种专门用于查看变量中存储内容的函数，上述代码分别使用 `head()` 函数查看了 x 的前 6 行，直接输入 x 或使用 `print()` 函数、`message()` 函数查看了 x 中的元素。最后一种方法是在创建 x 变量时在语句的最外边加上一组圆括号，也能令 R 返回代码执行的结果。此外，我们已经了解的查看 R 中变量信息的函数还有 `summary()` 函数、`fivenum()` 函数等。

对于有些函数来说，R 能够返回的信息很丰富，比如相关分析函数、回归分析函数与方差分析函数，但对于有些函数来说，R 返回的信息则很贫乏，特别是一些我们自己编写的函数往往只有一两个返回值。此时 `print()` 函数和 `message()` 函数能够帮助返回较

多的信息。

```
> f <- function(x) {
+ message(x)
+ print(x)
+ }
> f(c(1:5))
12345
[1] 1 2 3 4 5
```

上述代码创建了一个简单的函数 f ，它接受一个参数 x ，函数体由两句代码构成，分别使用 `message()` 函数和 `print()` 函数输出了 x 中的值。最后一行命令调用了 $f()$ 函数，并将参数 x 设置为 `c(1:5)`，R 返回了两行结果，其中第 1 行为 `message()` 函数对 x 的输出，第 2 行为 `print()` 函数对 x 的输出。

在函数中调用 `message()` 函数和 `print()` 函数是非常有用的，它能够帮助用户确切地知道函数中变量的变化情况或函数的更新状态，当函数非常复杂或运行时间过长时，这一点尤其有用。

```
> suppressMessages(f(c(1:5)))
[1] 1 2 3 4 5
```

通常来说，`message()` 函数的应用比 `print()` 函数更广泛。这是由于 `message()` 函数允许关闭其信息。上述代码将 $f()$ 函数作为 `suppressMessage()` 函数的传入对象调用了 $f()$ 函数，`suppressMessage()` 函数屏蔽了 `message()` 函数的返回结果，R 仅返回了 $f()$ 函数中 `print()` 函数给出的输出结果。

显然，`suppressMessage()` 函数用于 `message()` 函数重复输出大量相同代码时的情况，该函数允许我们仅在想要看到 `message()` 函数提供的信息时才会看到。

警告信息与正常情况下 R 返回的信息有些相似，它发生于程序出现了一些问题，但问题不至于严重到无法运行程序的时候。

```
> x <- readLines("warning.txt")
Warning message:
In readLines("warning.txt") : incomplete final line found on 'warning.txt'
> x
[1] "这是一个警告示例"
```

上述代码给出了一个关于警告的实例。在上述代码中，第 1 行命令使用 `readLines()` 函数按行读取了 `warning.txt` 中的信息，此时 R 返回了一个警告信息，提示我们在执行命令 `readLines("warning.txt")` 时出现了一些值得注意的事情，具体为 `warning.txt` 的最后一行不完整。

第 2 行命令查看了 x 中的内容，显然，尽管 R 给出了警告信息，但 x 还是成功读入了 `warning.txt` 中的内容，即文本内容“这是一个警告示例”。出现这一条警告信息的原因在于，在我们创建 `warning.txt` 时，在第一行输入文本“这是一个警告示例”后，并未换行，将光标移至下一行，而是将光标仍停留在第一行，因此，R 认为这行文本还未输入完毕。此时只需回车，将光标移至下一行，即可去掉 R 的警告信息。

16.1.2 R 代码中的警告处理方法

当 R 中出现警告信息时，通常意味着发生了一些不太好的事情。造成警告的常见原因有不合法的参数输入、糟糕的数值精度，或者一些分析人员事先没有考虑到的问题。警告信息表明程序可能已按照你所期望的那样执行，也可能并未如预料的那般正常工作。产生警告的原因多种多样，当然，最好的做法是找出原因，并完善它。

当产生警告的原因已知，并被确定无伤大雅，而完善它又不是特别方便时，`suppressWarnings()` 函数能够屏蔽警告信息。

```
> suppressWarnings(x1 <- readLines("warning.txt"))
> x1
[1] "这是一个警告示例"
```

上述代码在按行读取 `warning.txt` 时使用 `suppressWarnings()` 函数屏蔽了警告信息，再次查看 `x1` 中的内容，它与 `x` 中的内容并无不同。

关于警告信息的另一个有趣的特性是警告信息的出现次数。如下代码给出了与此相关的两个例子。

```
> for(i in 1:2){
+ x <- readLines("warning.txt")
+ }
Warning messages:
1: In readLines("warning.txt") :
  incomplete final line found on 'warning.txt'
2: In readLines("warning.txt") :
  incomplete final line found on 'warning.txt'
> for(i in 1:20){
+ x <- readLines("warning.txt")
+ }
There were 20 warnings (use warnings() to see them)
```

上述代码给出了两个 `for` 循环，在第一个 `for` 循环中，循环执行了两次，即按行读取了两次 `warning.txt` 文件，R 返回了两个一模一样的警告信息。在第二个 `for` 循环中，我们将循环次数增加为 20 次，此时 R 的返回结果变为了一个提示信息，提示存在 20 条警告信息，并告诉我们可以输入 `warnings()` 以查看这 20 条警告信息。实际上，当警告信息多于 10 条时，R 不会再直接给出这些信息，而是要我们使用 `warnings()` 来查看它们。

```
> getOption("warn")
[1] 0
```

全局选项 `warn` 是一个有关警告信息的重要选项，它用于确定警告的处理方法。`warn` 的默认值为 0，即执行当前代码，并报警。`getOption()` 函数能够查看 `warn` 的默认设置。

```
> options(warn=1)
> getOption("warn")
[1] 1
```

```
> x2 <- readLines("warning.txt")
Warning in readLines("warning.txt") :
  incomplete final line found on 'warning.txt'
> x2
[1] "这是一个警告示例"
```

当 `warn` 的值取为 1 时，R 对程序中的警告信息变得更加严苛，上述代码首先使用 `options()` 函数将 `warn` 设置为 1，然后重新按行读取了 `warning.txt`，R 同样返回了一个警告信息，`x2` 也成功读入了文本文件中的内容。但当程序中的警告信息多于 10 条时，R 不再仅提示存在一些警告信息，以及告诉我们可以使用 `warnings()` 查看，而会直接返回这些警告信息。

```
> options(warn=2)
> getOption("warn")
[1] 2
> x3 <- readLines("warning.txt")
Error in readLines("warning.txt") :
  (converted from warning) incomplete final line found on 'warning.txt'
> x3
Error: object 'x3' not found
```

当 `warn` 值取到大于 1 的值时，R 将更加严苛地对待程序中的警告信息，即将警告信息按照错误信息的标准来处理，不执行代码。上述代码首先将 `warn` 值设置为 2，并再次尝试按行读入 `warning.txt`，但此时 R 返回了一个错误信息，并表明这是一个由警告信息生成的错误信息。查看 `x3`，显然，并未成功创建该变量，也未成功读入文本信息。

与之类似的，`warn` 也能取为负数，当它的取值小于 0 时，R 将无视程序中的警告信息，不过这是一件非常危险的事情，通常并不建议用户这样做。

16.2 R 代码中的错误与错误处理方法

本节将讨论 R 代码中常见的错误信息与一些之前我们并未提到过的错误处理方法。通过阅读本节内容，读者将了解到在程序出现错误，或预知程序可能会出现错误时，如何尽量挽救它。

16.2.1 使用 `try()` 函数处理错误信息

错误信息是最为严重的一类 R 返回信息。当 R 报错时，就代表当前执行的命令具有无法补救的错误，R 必须停止它。但程序报错很常见，函数名不正确、参数名不正确、传入的参数具有不合法的形式、忘记添加分隔符、在错误的地方添加了错误的分隔符等，都是导致程序报错的常见原因。

```
> x <- readLines("1")
Error in file(con, "r") : cannot open the connection
```

```
In addition: Warning message:
In file(con, "r") : cannot open file '1': No such file or directory
```

上述代码是一个新手常见的错误命令，该命令使用 `readLines()` 函数按行读取文件“1”，但 R 给出了一个 Error，其具体提示信息为不能打开文件“1”，没有这个文件或字典。很明显，这里的问题在于向 `readLines()` 函数传入文档名时没有添加扩展名，在代码中将“1”修改为“1.txt”即能够正常打开文件。

这个错误很容易被发现，也很容易被纠正。但有时我们并不确定命令是否会出现错误，也不确定在这个对象上起作用的命令是否也能在其他对象上正确执行。一种可行的方法是在执行命令时使用 `try()` 函数调用命令行，并检查 `try()` 函数的返回结果。

```
> result <- try(x <- readLines("1"))
Error in file(con, "r") : cannot open the connection
In addition: Warning message:
In file(con, "r") : cannot open file '1': No such file or directory
```

上述代码在 `try()` 函数中传入了命令 `x <- readLines("1")`，并将 `try()` 函数的运行结果赋给了 `result`。`try()` 函数会尝试执行传入它其中的命令行，并返回执行结果。观察 R 的返回结果，R 同样返回了一个错误信息，实际上，`try()` 函数并不能将错误的命令改善为正确的命令，也不能令 R 明白这行命令的真实目的是去读取文件 1.txt，`try()` 函数的真正作用是记录命令是否能够正常运行。

```
> result
[1] "Error in file(con, \"r\") : cannot open the connection\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in file(con, "r"): cannot open the connection>
```

上述代码查看了 `result` 中的内容，它存储了有关命令行 `x <- readLines("1")` 的具体错误信息，此外，它还具备属性“try-error”，当 `try()` 函数执行的代码返回错误信息时，`try()` 函数将得到一个 `try-error` 类型的返回结果，显然，通过判断 `try()` 函数返回结果的类型，即可得知 `try()` 函数执行的命令能否正常执行，并根据判断结果对试执行的命令加以修补。

```
> if(inherits(result,"try-error")){
+ (x <- readLines("1.txt"))
+ }else{
+ print("result is not try-error")
+ }
[1] "这是文档1"
```

上述代码完善了试执行的命令。在上述代码中，`inherits()` 函数用于检查第一个参数的属性是否与第二个参数相同，并返回一个逻辑值，由于 `result` 的类型确实是 `try-error`，因此，此时 `if` 语句接受的逻辑值为 `TRUE`，流程控制语句进入第一条分支，重新适应

`readLines()` 函数读取了文件 “1.txt”，并赋给变量 `x`。这一行语句显然是对错误命令的弥补措施。

`else` 分支则给出了当 `try()` 函数试执行的语句能够正常执行时的措施，在这里令该措施为输入 “result is not try-error”。观察 R 的返回结果，此时显然进入了第一个分支，由于我们为第一个分支的语句加上了圆括号，故此处返回了 `x` 中的内容。

16.2.2 将 `try()` 函数与循环相结合

`try()` 函数提供了非常神奇的功能，它能够捕捉到执行的命令行是否能正常执行。显然，`try()` 函数只有和流程控制语句结合起来才有用武之地。

Error 的另一个令人烦恼的特性是它会终止全部程序，在循环调用命令时，这可能带来大麻烦，比如，在读取一批文件时，任何一个文件不能正常读取都会导致程序报错，即所有文件都无法正常读取。`try()` 函数提供了一种补救的可能，即在循环语句中保留能正常执行的命令生成的结果。

```
> for(i in 4:1){
+ result <- try(x <- readLines(paste(i, ".txt", sep="")))
+ if(inherits(result, "try-error")){}
+ else{print(x)}
+ }
Error in file(con, "r") : cannot open the connection
In addition: Warning message:
In file(con, "r") : cannot open file '4.txt': No such file or directory
[1] "这是文档3"
[1] "这是文档2"
[1] "这是文档1"
```

上述代码实现了一条 `for` 循环语句。循环体一共循环 4 次，在这 4 次循环中，`i` 的值分别取为 4、3、2、1。循环体包含三行语句，第 1 行语句使用 `try()` 函数执行了命令 `x <- readLines(paste(i, ".txt", sep=""))`，并将 `try()` 函数的结果保存在 `result` 中，其中 `paste()` 函数起到连接作用，它连接了 `i` 和 `.txt` 两部分，参数 `sep` 表明连接此二者时不留任何空格，随着 `i` 的变化，在 4 次循环中，`readLines()` 函数的读取对象分别为 `4.txt`、`3.txt`、`2.txt`、`1.txt`。

第 2、3 行语句为一组流程控制语句，当 `result` 包含属性 `try-error` 时，不执行任何命令，否则输出 `x`。在本例中，R 的工作目录下仅包含 `3.txt`、`2.txt`、`1.txt` 这三个文本文件，并没有 `4.txt`，故 `readLines()` 函数在读取 `4.txt` 时将会报错，`result` 将包含属性 `try-error`，流程控制语句也将进入 `if` 分支。

观察 R 的返回结果，它首先返回了一条错误信息，在第一次循环中，`try()` 函数试执行的命令未能成功执行，故出现了错误信息。接下来的三条语句则是第 2、3、4 次循环中成功读取文件并进入 `else` 分支后输出的结果。显然，`3.txt`、`2.txt`、`1.txt` 中存放的内容分别为“这是文档 3”、“这是文档 2”和“这是文档 1”。

```
> txt <- c("4.txt", "3.txt", "2.txt", "1.txt")
> txt <- as.list(txt)
```

```

> library(plyr)
> tryapply(txt,function(x){readLines(x)})
[[1]]
[1] "这是文档3"

[[2]]
[1] "这是文档2"

[[3]]
[1] "这是文档1"

Warning message:
In file(con, "r") : cannot open file '4.txt': No such file or directory

```

注意，在使用 `try()` 函数时，错误信息和正常返回的内容是交错出现的，`tryapply()` 函数能够从返回内容中去掉错误信息，这能够美化代码。

上述代码首先创建了一个向量 `txt`，其中存放了 4 个准备读入的 `txt` 文件名。`tryapply()` 函数接受列表形式的输入，因此第 2 行代码将 `txt` 转换为了列表。第 3 行代码加载了 `plyr` 程序包，第 4 行代码则调用了 `plyr` 包中的 `tryapply()` 函数。

`tryapply()` 函数接受如下两个参数，第一个参数为列表形式的被应用对象，第二个参数则为被应用函数，`tryapply()` 函数将对列表中的每一个元素逐一应用该函数。在上述代码中，我们将被应用函数设置为按行读取列表中的每一元素。

执行上述代码，R 返回了一个列表形式的结果，前三个元素是读取 `3.txt`、`2.txt`、`1.txt` 后返回的结果，最后则返回了一个警告信息，提示函数中有未能成功执行的部分。与上一个例子的结果相比，此时错误信息确实被挪到了最后，得到了最佳的输出效果。

16.3 调试 R 代码

在前两节中讨论了 R 的三种信息，并讨论了它们各自代表的意义。本节将讨论如何利用 R 提供的函数找出代码中的错误，并纠正它们。本节的内容能帮助读者写出优秀、通用的代码，并指导读者在代码报错时使用最方便的方法调试它们。

16.3.1 查看调用栈或暂停代码

R 为用户提供了一些查看全部变量信息的函数，在它们的帮助下，代码中大部分错误都能够很容易被找出来，但当错误出现在非常复杂的函数嵌套调用中时，要确定错误发生的位置则变得困难了许多。对于这种情况，`traceback()` 函数提供了一种寻找错误的方法。

```

> bar <- function(x) {x+2}
> foo <- function(x) {bar(x)}
> foo(a)
Error in bar(x) : object 'a' not found

```

```
> traceback()
2: bar(x) at #1
1: foo(a)
```

上述代码创建了如下两个函数，第一个函数 `bar` 接受一个参数 `x`，并计算了 `x` 与 2 的和；第二个函数 `foo` 也接受一个参数 `x`，并将 `x` 传入函数 `bar` 中，调用了函数 `bar`，显然，这里完成了一次函数的嵌套。第 3 行命令调用了函数 `foo`，并将 `a` 作为参数值传入 `foo`。

R 返回了一个错误信息，其具体内容为找不到对象 `a`。这个提示信息是比较模糊的，查看我们创建的函数 `foo` 和函数 `bar`，显然，函数 `bar` 需要一个数值型的参数，这样才可以与 2 作和，但 `a` 显然不是我们需要的数值型参数，因此 R 会报错。

在嵌套函数结构简单时，我们能够直接看出错误出现在哪里。但当函数嵌套层数过多，或函数体比较复杂，或嵌套函数和循环函数组合起来构成一个无比复杂的代码时，`traceback()` 函数就派上了用场。它能够沿着调用栈向上追溯，直到找出出现错误的函数为止。

上述代码的最后一行输入了命令 `traceback()`，R 返回了两个调用层。第一个调用层为 `foo(a)`，当 `a` 作为参数 `x` 的值传入 `foo()` 函数时代码还能正常执行；第二个调用层为 `bar(x)`，当调用到 `bar()` 函数时代码出现了错误，这表明代码的错误出现于 `foo()` 函数内的 `bar()` 函数处。显然，当代码中函数被来回调用时，通过调用栈追查错误有助于明确错误出现的具体位置，这对解决错误很有帮助。

```
> bar <- function(x) {
+   browser()
+   x+2
+ }
> foo <- function(x){bar(x)}
> foo(a)
Called from: bar(x)
Browse[1]>
debug at #3: x + 2
Browse[2]>
Error in bar(x) : object 'a' not found
```

当你要编写一个复杂的函数，而不能保证该函数一定能成功执行时，在代码中添加 `browser()` 函数是一种可行的选择。`browser()` 函数起到暂停程序的作用，R 在读取到 `browser()` 函数后将暂停执行程序，并逐行执行后续代码，这相当于慢动作地执行代码，让用户能够看清楚错误到底出现在哪里。

上述代码同样创建了 `bar()` 函数和 `foo()` 函数，并在 `bar()` 函数的函数体前添加了命令 `browser()`。第 3 行命令将 `a` 传入了 `foo()` 函数。接下来发生了一些较为复杂的情况。R 首先返回了提示命令“Called from: bar(x)”，然后返回了命令“Browse[1]>”，在这里，R 将有一个暂停。很明显，此时 `foo(a)` 已经将 `a` 传入了 `bar()` 函数，`bar()` 函数也运行到了命令 `browser()` 处，到现在为止，R 还未报错。

在“Browse[1]>”处，用户可以输入一些命令，但这些命令的结果并不会保留在 R 环境中。

比如，此处可以输入 `ls.str()` 以查看目前都有哪些变量被创建成功。但在这里创建一个新变量则毫无用处，比如在这里创建一个数值型变量 `a` 并不能使 `a` 保留下来。

如果在 “`Browse[1]>`” 处什么也不做，直接按下回车键，R 将开始执行下一行。在本例中，即命令行 “`x+2`”，R 也将返回一个提示目前函数执行的具体位置，以及命令行 “`Browse[2]>`”。如果该行命令能够正常执行，并且还有后续代码，那么按下回车键后 R 将执行下一行命令；如果该行命令有错，那么 R 将报错，并直接跳出代码，就像本例所示的那样。

16.3.2 修改 error 选项

与警告信息类似，错误信息也有其全局选项 `error`，它的默认定义是一个函数类，使用 `getOption()` 函数能够查看其详细定义。`error` 也能够设置为其他函数，比如自己创建的一些函数，或一些 R 自带的函数，其中最常用的是 `recover()` 函数，它能够按行调试代码。

```
> bar <- function(x){x+2}
> foo <- function(x){bar(x)}
> options(error=recover)
> foo(a)
Error in bar(x) : object 'a' not found

Enter a frame number, or 0 to exit

1: foo(a)
2: #1: bar(x)

Selection:
```

上述代码首先创建了 `bar()` 函数和 `foo()` 函数，它们均接受一个形参 `x`，其中 `bar()` 函数将 `x` 与 2 相加，`foo()` 函数则调用了 `bar()` 函数。第 3 行命令使用 `options()` 函数将 `error` 的值修改为 `recover`，再次强调，`recover` 是一个函数，它能够实现按行调试代码的功能。

第 4 行命令调用了 `foo()` 函数，并将 `a` 作为参数传入了 `foo()`，R 首先返回了一条错误信息，表明该代码不能正常执行；第 2 行信息提示用户输入一个调用层的层数，或输入 0 以退出调试过程；接下来的第 3、4 行信息则给出了直到错误发生之前进入过的调用层，这两个调用层在上文中有关 `traceback()` 函数的部分已经介绍过。

最后一行信息 “`Selection:`” 则给出了交互接口。

```
Selection: 2
Called from: top level
Browse[2]>

Enter a frame number, or 0 to exit

1: foo(a)
2: #3: bar(x)
```

```
Selection: 0
```

上述代码紧接上一个例子，首先在“Selection:”后输入了数字 2，此时 R 返回了正在调试的程序的第二个调用层，并给出了另一个交互接口“Browse[2]>”，这一部分与介绍 `browser()` 命令时提到的“Browse[2]>”相一致，我们同样可以在这里查看此时生成的变量，但此处按下回车键后，R 并不会移至下一行命令，而是返回“Selection:”交互接口处，用户可以选择再次进入某一调用层或退出调试过程。

```
> debug(foo)
> foo(a)
debugging in: foo(a)
debug at #1: {
  bar(x)
}
Browse[2]>
debug at #1: bar(x)
Browse[2]>
Error in bar(x) : object 'a' not found
```

另一个能够实现按行调试代码的函数是 `debug()` 函数，它也能够按行调试代码。上述代码首先向 `debug()` 函数中传入 `foo`，表明对 `foo()` 函数应用 `debug()` 函数。第 2 行命令调用了 `foo()` 函数，并将 `a` 作为参数传入 `foo()`。

R 返回的结果与 `browser()` 命令返回的结果较为相似，它同样逐行逐行地暂停，并在每次暂停时给出交互接口“Browse[2]>”以便用户输入命令查看当前的 R 环境。`debug()` 函数同样也在程序报错时便立刻退出调试过程。

修改 `error` 的默认值或设置 `debug()` 函数都将产生长远的影响，`error` 的默认值被修改为 `recover` 后，接下来每次调用函数出错时都会进入调试过程，而在 `debug()` 函数中传入某一函数后，接下来每次调用该函数也都会进入调试过程，在上例中，即意味着每次调用 `foo()` 函数时都会调试 `foo()`。这种影响也可以叠加，即同时对一个函数执行两种调试方法，只有当退出 R 时，这种影响才会消失。

16.4 向量化编程方法

本节将讨论如何编写占用资源更少、运行更快速的函数。通过阅读这部分内容，读者将对 R 的工作方式有进一步的了解，并明白如何优化自己的代码。

16.4.1 向量化编程思想

R 是一种高级语言，在执行某段 R 代码时，R 需要首先将 R 语言转换为某种计算机能明白的更低级的语言，计算机利用底层语言运行这些代码，并生成结果，然后再将结果传回 R。

对于非向量化的程序，比如 `for` 循环或 `while` 循环，R 将第一个元素调入内存，转换为低级语言，将结果传回 R；将第二个元素调入内存，转换为低级语言，将结果传回 R……显然，在转换语言与传回结果时，非向量化的程序浪费了很多时间。而向量化的程序，比如 `apply` 函数族，则将全部元素一起调入内存，转换为低级语言后计算结果，并将结果一次性传回 R，这将节约许多时间。

```
> s <- 0
> for(i in 1:10){s <- s+i^2}
> s
[1] 385
> t <- sum((1:10)^2)
> t
[1] 385
```

上述代码首先给出了一个 `for` 循环，它的循环变量 i 从 1 循环到 10，在每次循环时都令 s 叠加了 i 的平方，查看 s 的值，385 显然为数值 1 到 10 的平方和。

对于为数值 1 到 10 求平方和的例子来说，`sum()` 函数能提供向量化的解决方法。上述代码同样成功求出了 1 到 10 的平方和，使用 `sum()` 函数具有更简洁的命令输入方式，也能够更迅速地执行程序。

```
> colMeans(cars)
speed  dist
15.40 42.98
> colSums(cars)
speed  dist
770 2149
> pmin(c(1,2,3),c(3,2,1))
[1] 1 2 1
> pmax(c(1,2,3),c(3,2,1))
[1] 3 2 3
```

R 中的大部分函数都支持向量化，但有些函数则单独提供一个变形后的非向量化的函数。上述代码给出了 4 个向量化的函数 `colMeans()`、`colSums()`、`pmin()` 和 `pmax()`，它们分别是 `mean()` 函数、`sum()` 函数、`min()` 函数和 `max()` 函数向量化的版本。其中，`colMeans()` 函数能够对数据框按列求均值，类似地，还有能够对数据框按行求均值的 `rowMeans()` 函数；`colSums()` 函数能够对数据框按列求和，类似地，还有能够对数据框按行求和的 `rowSums()` 函数。而 `pmin()` 函数与 `pmax()` 函数则分别求出了几个向量的最小值和最大值。

16.4.2 比较循环和向量的运行速度

通常来说，向量化编程的运行时间要比非向量化编程的运行时间要少一两个数量级。在 R 中，解决一个问题的可行方法往往有好几种，不妨来实际比较一下不同方法的运行速度。

```
> s <- 0
> system.time(
+ for(i in 1:1e+06){s <- s+i^2}
+ )
用户 系统 流逝
1.69 0.00 1.70
> system.time(t <- sum((1:1e+06)^2))
用户 系统 流逝
0.06 0.01 0.08
```

`system.time()` 函数能够记录命令运行的时间。上述代码分别记录了使用循环计算一列数列的平方和的运行速度，以及使用 `sum()` 函数计算一列数列的平方和的运行速度。为了更明确地比较两种运行速度的差异，上述代码计算了 1~1 000 000 这 10 000 000 个数值的平方和。

观察 R 的返回结果，使用 `for` 循环计算数列的平方和时，总耗时为 3.39 秒，使用 R 自带的向量化函数 `sum()` 计算数列的平方和时，总耗时为 0.15 秒，此二者恰好相差一个数量级，很明显，`sum()` 函数的运行速度要比 `for` 循环快得多。

```
> x <- as.matrix(1:1e+06)
> head(x)
      [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
> fun <- function(x){
+ y <- as.numeric(x)
+ s <- s+y
+ }
> s <- 0
> system.time(u <- apply(x,2,fun))
用户 系统 流逝
0.12 0.00 0.12
```

`apply` 函数族是另一种常用的向量化函数，上述代码检测了 `apply()` 函数在求和时的运行速度。`apply()` 函数接受一个数据框作为函数应用对象，上述代码首先创建了一个仅包含一行数据的数据框 `x`，它存储了 1 000 000 个数值。

接下来的 4 行代码创建了函数 `fun`，它接受一个参数 `x`，将 `x` 转化为数值型变量 `y`，并将 `y` 的值叠加到 `s` 上。第 7 行代码将变量 `s` 重置为 0，第 8 行代码调用 `apply()` 函数对 `x` 按列求和，即求得了 1~1 000 000 的和，并使用 `system.time()` 函数记录了命令执行的时间。

R 返回了三个运行时间，`apply()` 函数总的执行时间为 0.24 秒，这一速度比 `for` 循环要快，但比 `sum()` 函数要慢。实际上，`apply()` 函数内部同样使用 `for` 循环实现，它的执行速度与执行一次 `for` 循环基本相同。

```
> s <- 0
> system.time(
+ for(i in 1 :1e+06){
+ for(j in 1 :1){
+ s <- s+x[i,j]
+ }
+ }
+ )
用户 系统 流逝
3.30 0.00 3.34
```

但对于执行多次 for 循环来说，`apply()` 函数的速度要快一些。在上述代码中，使用两组 for 循环实现了与 `apply()` 函数相同的功能，即对数据框中的数据求和。第一组循环中循环变量 i 从 1 循环至 1 000 000，对应着 x 中的 1 000 000 行数据；第二组循环中循环变量 j 只有一个取值 1，对应着 x 中的一列数据。 s 同样为 1~1 000 000 全部数值的和。

观察 R 的返回结果，嵌套调用两个 for 循环时，系统的运行时间达到了 6.64 秒，这与 `apply()` 函数的运行时间几乎要差出两个数量级。显然，循环函数的嵌套调用带来的转换时间会呈指数倍增长，而 `apply()` 函数则可以尽可能地节约转换时间，即将两个嵌套的循环过程简化为一个循环。

第 17 章 构建电影评分预测模型

电影评分预测系统是推荐系统的一种，本章使用 R 构造了一个较为简单的评分预测模型。本章使用较多的 R 程序包，详细介绍有关原始数据处理方法的基本知识，同时也完成了一次模型的构建和评估。通过阅读本章，读者将首次接触到一个完整的模型，此外，读者也将学习到有关协同过滤的知识。

17.1 获取数据并探索

电影评分如今被广泛应用在电影行业中，无论是去电影院看电影，还是在在线网站上看电影，它都是一个选择电影的重要参考。如果投资商能够在电影未拍摄前就预测到新电影的评分，那么就可以使电影投资的回报最大化，而且观众也将看到更多的好电影。预测某个具体用户对新电影的评分也可以使网站更准确地向其做出电影推荐。

本章的主题就是构建一个电影评分预测模型。我们使用来自 <http://grouplens.org/datasets/movielens/> 网站的数据，该网站提供了好几种不同数量级的数据，我们选择了数据量最小的一份数据，这份数据包含约 1 000 人对约 1 700 部电影做出的 100 000 次评分，一共有 1、2、3、4、5 五种评分分数，分数越高，表示该用户对该电影越欣赏。

下载好数据后，首先需要将数据放到 R 的工作目录下，只有这样才能方便地读取数据。如果不清楚 R 的当前工作目录，可以使用 `getwd()` 命令查看，也可以使用 `setwd()` 命令将 R 的当前工作目录修改到存放了数据的位置，不过一般不这样做。

如下是向 R 中载入数据的代码：

```
> ml100k <- read.table("u.data", header=F, stringsAsFactors=T)
> head(ml100k)
  V1  V2 V3      V4
1 196 242  3 881250949
2 186 302  3 891717742
3  22 377  1 878887116
4 244  51  2 880606923
5 166 346  1 886397596
6 298 474  4 884182806
```

首先，第 1 行代码使用 `read.table` 函数将“u.data”文件（下载的电影数据文件）中的数据读入 `ml100k` 中，参数 `header=F` 指定不读取表头，`stringsAsFactors=T` 则指定表中所有的列都不是因子，也就是数值型数据。

向 R 中读取完数据后，使用 `head()` 命令读取了 `ml100k` 的前 6 条数据。u.data 文件

中有近万条数据，将其全部读取到 R 屏幕上是非常不现实的，`head()` 命令则可以很方便地查看数据表的内容。

由 R 返回的数据可以看到 `ml100k` 由 4 列数据构成，其中第 1 列数据是用户 ID，第 2 列数据是电影 ID，第 3 列数据是该用户对该电影的评分，第 4 列数据是用户评论的时间。这 4 列数据中第 4 列数据是没有用处的，下一步要做的就是将其删除，为了删除列数据，同样需要执行两条代码：

```
> ml100k <- ml100k[,-4]
> head(ml100k)
  V1  V2 V3
1 196 242  3
2 186 302  3
3  22 377  1
4 244  51  2
5 166 346  1
6 298 474  4
```

在上述两行命令中，第 1 行命令中的 `ml100k[,-4]` 是一个常见的矩阵表示，其中逗号将 `ml100k` 的行和列分隔开来，由于这里仅删除第 4 列，而不改变行信息，因此逗号前不添加信息，逗号后使用“-4”删除第 4 行数据。第 2 行命令则再次读取了 `ml100k` 中的前 6 行数据。由 R 返回的数据可知，此时第一条命令已经达到我们想要的效果。

在此时 `ml100k` 中剩余的三列数据中，`V1` 和 `V2` 是关于用户和电影的 ID 号，而 `V3` 则是具体的电影评分。由于我们的目的是预测电影评分，因此 `V3` 无疑是最具有价值的。而 `V3` 是一个分布在 1~5 之间的离散数据，因此探索其极值、最值的意义都不太大，直方图则可以精确地概括出 `V3` 的分布特点。有关绘制直方图的代码如下：

```
> library(ggplot2)
> ggplot(ml100k, aes(x=V3)) + geom_histogram(binwidth=0.01)
```

R 中有许多绘制直方图的函数，我们选择使用 `ggplot2` 函数包中的函数。为了调用 `ggplot2` 包中的函数，首先需要使用 `library()` 命令载入该函数包，注意，`library()` 命令中的函数包名称不需要用引号括起来。

第 2 行命令中 `ggplot(ml100k, aes(x=V3))` 说明 `ml100k` 数据表将作为作图函数的数据来源，`aes(x=V3)` 则指定数据列 `V3` 作为作图函数的 `x` 轴参数；`geom_histogram(binwidth=0.01)` 则画出了一个列宽为 0.01 的直方图。

执行上述两条代码，便可得到如图 17.1 所示的直方图。由于电影评分有 5 个等级，因此 `x` 轴也由坐标 1、2、3、4、5 组成，这里并未对 `y` 轴定义，因此 `y` 轴仅对每个等级进行计数。

由图 17.1 可知，用户给出三星评分和四星评分的情况最多，给出一星评分和二星评分的情况最少。其中三星评分、四星评分的个数加起来占全部评分的 75% 左右。用户关于电影的评分呈较明显的左偏分布，这提示我们用户对电影的评分通常略高，新电影的评分低于 3.5 时就已经失去了一半观众。

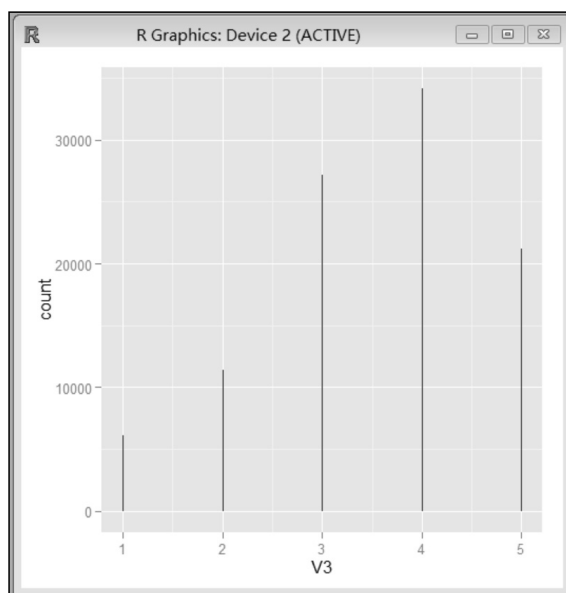


图 17.1 histogram 函数形成的直方图

17.2 利用 recommenderlab 包处理数据

在 17.1 节中，我们初步清洗并探索了数据，但数据仍未处理完全。在 17.1 节的最后，ml100k 仍是一个由三个变量、近万条数据构成的数据表，表中的每一条数据都单独记录了某个用户对某个电影的评分。这种形式其实并不方便分析，我们感兴趣的是以用户为行、电影为列的巨大评分矩阵。

图 17.2 所示为以用户为行、电影为列的评分矩阵，矩阵中给出了三个用户对 6 部电影的评分。我们接下来所要做的就是将 ml100k 数据表转换为类似图 17.2 的矩阵。为了将数据表转换为矩阵，需执行如下代码：

```
> library(reshape)
> ml100k <- cast(ml100k, V1~V2, value="V3")
> ml100k[1:3,1:6]
  V1  1  2  3  4  5
1  1  5  3  4  3  3
2  2  4 NA NA NA NA
3  3 NA NA NA NA NA
```

	A	B	C	D	E	F
1	2	3	4	4	3	4
2	3	3	2	2	4	2
3	1	2	3	4	2	2

图 17.2 以用户为行、电影为列的评分矩阵

reshape 函数包是一个专门转换数据格式的包,使用 reshape 包中的 cast 函数转换数据。上述代码首先载入 reshape 包,然后利用 cast 函数将 ml100k 转换成一个矩阵,cast 函数给出了三个参数,参数 ml100k 表示 cast 函数的数据来源,V1~V2 表示 cast 函数生成的新矩阵以 V1 为行、V2 为列,value=“V3”则表示使用 V3 的值填充矩阵。

如果 cast 函数运转正常,则 ml100k 已经变成一个与图 17.2 相似的电影评分矩阵,只是它的行和列都达到了上千条。由于现在的 ml100k 中每一条数据都包含上千个评分,因此,此时 head() 函数不再适合观察矩阵,使用命令 ml100k[1:3,1:6] 来查看 ml100k 中前三行、前 6 列数据。

```
> ml100k <- ml100k[,-1]
> ml100k[1:3,1:6]
  1  2  3  4  5  6
1  5  3  4  3  3  5
2  4 NA NA NA NA NA
3 NA NA NA NA NA NA
```

此时 R 返回的矩阵中第一列数据代表了每一条数据的序号,这条数据是我们所不需要的,因此执行第 4 条命令删去了 ml100k 中的第 1 列数据,再次执行第 3 条命令,如 R 的返回结果所示,ml100k 已经是我们所想要的巨型电影评分矩阵。

将 ml100k 转化为评分矩阵后,还需进一步修改 ml100k 的属性。我们希望利用 recommenderlab 包中有关协同过滤的函数构建模型,就需要首先将数据转化为 recommenderlab 包所能使用的类型。因此,我们需要观察并修改 ml100k 的类型属性。如下是查看并修改数据类型的代码:

```
> class(ml100k)
[1] "cast_df"      "data.frame"
> class(ml100k) <- "data.frame"
```

首先执行命令 class(ml100k),R 将返回 cast_df 和 data.frame,这代表 ml100k 有两个属性,分别是 cast_df 和 data.frame,其中 cast_df 属性是我们不需要的,因此执行第 2 条命令,将 data.frame 属性赋给 ml100k 矩阵,即只保留 ml100k 矩阵的 data.frame 属性。可以再次执行第 1 条命令,查看此时 ml100k 是否只剩下 data.frame 属性。

data.frame 属性固然是一个应用广泛的重要属性,但这还不是我们的终点,我们的目标是将 ml100k 转化成 recommenderlab 包所要求的 realRatingMatrix 属性。如下是利用 recommenderlab 包修改数据属性的命令:

```
> library(recommenderlab)
> ml100ka <- as.matrix(ml100k)
> ml100kb <- as(ml100ka,"realRatingMatrix")
> ml100kb
943 x 1682 rating matrix of class 'realRatingMatrix' with 99548 ratings.
```

必须首先执行 library(recommenderlab) 命令以加载 recommenderlab 包,否则 R 将无法辨识 realRatingMatrix 属性。然后使用 as.matrix() 函数将 ml100k 由 data.frame 类型转化

为 `matrix` 类型，并赋给 `ml100ka`，由于 `ml100k` 中只有数值型变量，因此这一步转换非常顺利。然后又利用 `as()` 函数强制将 `ml100ka` 由 `matrix` 类型转化为 `realRatingMatrix` 类型，并赋给 `ml100kb`。

最后，再次执行命令 `ml100kb`，R 将返回 `ml100kb` 的属性：

```
943 × 1682 rating matrix of class 'realRatingMatrix' with 100000 ratings.
```

即由 943 行、1 682 列 `realRatingMatrix` 类型的数据构成的矩阵，它包含 100 000 条评分。

17.3 建立模型并评估

在 17.2 节中，我们已经完成了全部的数据预处理，本节将要正式开始建模。本节将比较几种不同模型的特点，并演示如何使用 `recommenderlab` 包建立模型，在电影评分预测这一问题上，本节还将根据几种指标比较不同模型的优劣。

17.3.1 模型的选择与建立

针对 `realRatingMatrix` 数据类型，`recommenderlab` 包一共提供了 6 种模型，它们分别是基于项目协同过滤（IBCF）、主成分分析（PCA）、基于流行度推荐（POPULAR）、随机推荐（RANDOM）、奇异值分解（SVD）和基于用户协同过滤（UBCF）。

在 17.2 小节中，我们已创建了电影评分矩阵 `ml100k`，同时也查看它的前三行、前 6 列数据，在这 18 个数据中有 11 个是空缺的。这是由于一个正常用户绝不可能将 1 700 部电影都看过，每个用户最多能够为几十部影片打分。因此，评分矩阵将是一个非常稀疏、含有许多空缺值的矩阵。

协同过滤主要分为两个步骤，首先依据目标用户的已知电影评分找到与目标用户观影风格相似的用户群，然后计算该用户群对其他电影的评分，并作为目标用户的预测评分。评分矩阵的稀疏性并不影响协同过滤的工作效果，因此，协同过滤较为适合建立评分预测模型。

在正式建模之前，还需要为 `ml100kb` 中的每一列数据命名，否则模型就会报错。执行如下两行命令：

```
> colnames(ml100kb) <- paste("M",1:1682, sep="")
> as(ml100kb, "matrix")[1:3,1:6]
  M1 M2 M3 M4 M5 M6
1  5  3  4  3  3  5
2  4 NA NA NA NA NA
3 NA NA NA NA NA NA
```

第 1 行命令将 `ml100kb` 中的 1 682 列数据命名为 M1、M2、M3、…、M1682，分别代表 1 682 部电影。此时可以执行 `as()` 命令来查看 `ml100kb` 中的数据。

第二行命令中 `as()` 函数指定以 `matrix` 格式查看 `ml100kb` 中前三行、前 6 列的数据，观察 R 返回的结果，可以看到此时每列数据的名称已变为我们所需要的形式。`as()` 函数还可以以 `list` 格式查看 `realRatingMatrix` 类型数据表中的数据，这一点在下文中也会提及。

完成了上述预处理工作，现在即可尝试建立第一个模型。不妨首先尝试用 UBCF 方法构建模型并预测评分：

```
> ml100k.model <- Recommender(ml100kb[1:800], method="UBCF")
> ml.predict <- predict(ml100k.model, ml100kb[801:803], type="ratings")
> as(ml.predict, "matrix")[1:3,1:6]
```

	M1	M2	M3	M4	M5	M6
[1,]	4.023833	4.017790	4.099041	4.061437	4.038462	4.038462
[2,]	3.719220	3.505469	3.482577	3.485396	3.373351	3.493333
[3,]	3.021637	3.090909	3.099141	3.099141	3.090909	3.090909

预测电影评分需要三行命令。第 1 行命令中的 `Recommender()` 函数包含如下两个参数，第一个参数指定使用 `ml100kb` 数据表中的前 800 条数据建模，第二个参数则指定建模方法为基于用户的协同过滤，此外，也可以指定 IBCF、PCA、POPULAR、RANDOM、SVD 等方法为建模方法。

第 2 行命令使用 `predict()` 函数进行预测，该函数包含三个参数，第一个参数指定 `ml100k.model` 为预测评分所使用的模型，第二个参数指定对 `ml100kb` 中的第 801~第 803 个用户进行预测，第三个参数指定预测类型为“ratings”，也就是预测这三个用户对 1 682 部电影的评分。

第 3 行数据则利用 `as()` 函数以 `matrix` 格式查看了 `ml.predict` 中的前三行、前 6 列数据。观察由 R 返回的数据，可以看到电影预测评分都落在 3~5 之间，这与原始数据集中分布在 3、4、5 上是吻合的。`predict()` 函数不仅可以预测电影评分，同时也可以为用户推荐电影，只需将 `predict()` 函数中的第三个参数稍作改变：

```
> ml.predict2 <- predict(ml100k.model, ml100kb[801:803], n=5)
> as(ml.predict2, "list")
```

```
[[1]]
[1] "M272" "M258" "M315" "M327" "M298"
```

```
[[2]]
[1] "M313" "M50" "M298" "M328" "M127"
```

```
[[3]]
[1] "M302" "M268" "M272" "M313" "M9"
```

为用户推荐电影是对预测评分的延伸应用。容易理解，如果某个用户对某部电影的预测评分较高，就意味着该用户喜爱这部电影的可能性较大，则向该用户推荐这部电影比较合适。

在上述代码中，首先将 `type="ratings"` 改为了 `n=5`，由于 `predict()` 函数的默认预测类型即为 `topNList` 类型，因此无须再次指定参数 `type`，只需指定 `n=5`，表示为每个用户推荐最佳的 5 部影片即可。

同时，第 2 行命令利用 `as()` 函数以 `list` 格式查看了 `ml.predict2` 中的数据，在 `ml.predict2` 中存储的是 `ml100kb` 中第 801~ 第 803 个用户最可能喜欢的 5 部电影，R 按照列表形式返回了这些电影的名称。

由 `predict()` 函数的两种用途可以看出，`recommenderlab` 包在处理电影评分数据上具有得天独厚的优势，在预测电影评分时，该包无疑是优先选择。不过 `recommenderlab` 包对于输入模型的数据类型有特殊要求，因此，熟练转换数据类型是使用这个包的必需知识。

17.3.2 模型之间的比较和评估

上文使用 `ml100kb` 中的数据建立了一个基于用户的协同过滤系统，并使用该系统完成了预测和推荐。但是一个完整的建模过程不但包含模型的建立，也包含模型的评估。在 `recommenderlab` 包所提供的 6 种模型中，我们感兴趣的是，究竟哪种模型的评分预测准确度最高呢？

为了回答这个问题，我们需要构造多个预测模型，并确定出评价模型好坏的参数，从而比较不同模型之间的优劣。如下 6 行代码一共构建了 5 个预测模型：

```
> model.eval <- evaluationScheme(ml100kb[1:943], method="split",
train=0.9, given=15, goodRating=4)
> model.random <- Recommender(getData(model.eval, "train"),
method="RANDOM")
> model.ubcf <- Recommender(getData(model.eval, "train"), method="UBCF")
> model.ibcf <- Recommender(getData(model.eval, "train"), method="IBCF")
> model.svd <- Recommender(getData(model.eval, "train"), method="SVD")
```

模型的评估是一个很复杂的问题。首先，需要将原始数据分为训练集和测试集两部分，通常情况下训练集要远远大于测试集。其次，测试集还需要进一步拆分为 `know` 和 `unknown` 两部分，其中 `know` 部分存放了用户已评价的电影 ID 和对应评价者的用户 ID，`unknown` 部分则存放了真实的评分。

当模型使用训练集中的数据训练完毕后，令模型对 `know` 部分中存放的 ID 号进行预测，并将预测结果与 `unknown` 部分的真实值相对比，模型对 `unknown` 部分预测的准确度将成为评价模型优劣的标准。尽管整个评估过程十分复杂，不过幸好 `recommenderlab` 包提供了 `evaluationScheme()` 函数。

`evaluationScheme()` 函数是一个专门用来设计评估方案的函数，我们在构建预测模型的首行命令中使用了它，并设置了 5 个参数。第一个参数指定模型数据集是 `ml100kb` 中的全体数据，第二个参数指定模型评估时将训练集和测试集分开验证，第三个参数指定抽取数据集中的 90% 作为训练集，第四个参数则指定测试集中的 `know` 部分的项目个数，最后一个参数表示预测成功的最小评分。

确定好各个数据集的大小后，便可以使用 `Recommender()` 函数来训练模型。由于主成分分析需要多个变量，因此在电影评分预测问题中，我们仅构造了其他 5 种模型。这 5 种模型均使用 `getData()` 函数将 `model.eval` 中的 `train` 部分作为数据集传入 `Recommender()` 函数中：

```

> predict.random <- predict(model.random, getData(model.eval, "known"),
type="ratings")
> predict.ubcf <- predict(model.ubcf, getData(model.eval, "known"),
type="ratings")
> predict.ibcf <- predict(model.ibcf, getData(model.eval, "known"),
type="ratings")
> predict.pop <- predict(model.pop, getData(model.eval, "known"),
type="ratings")
> predict.svd <- predict(model.svd, getData(model.eval, "known"),
type="ratings")

```

上述 5 行代码使用 `predict()` 函数对 Known 部分的数据进行预测，之前训练得到的 5 个模型分别作为 5 个 `predict()` 函数的输入模型，并且使用 `getData()` 函数将 `model.eval` 中的 `know` 中存放的 ID 数据作为预测目标，预测它们的评分。

`recommenderlab` 包同样提供了度量预测结果准确度的函数。`calcPredictionAccuracy()` 函数能够计算两组数据的均方根误差 (RESM)、均方误差 (MSE) 和平均绝对误差 (MAE)，这三种误差都是用来度量两组数据的相似程度的，误差越小，预测结果与真实结果就越接近，模型的预测效果也就越好。如下是计算误差的代码：

```

> error <- rbind(calcPredictionAccuracy(predict.random, getData(model.
eval, "unknown")),
+ calcPredictionAccuracy(predict.ubcf, getData(model.eval, "unknown")),
+ calcPredictionAccuracy(predict.ibcf, getData(model.eval, "unknown")),
+ calcPredictionAccuracy(predict.pop, getData(model.eval, "unknown")),
+ calcPredictionAccuracy(predict.svd, getData(model.eval, "unknown")))
> error

```

	RMSE	MSE	MAE
[1,]	2.111648	4.459057	1.7232781
[2,]	1.025763	1.052190	0.8146407
[3,]	1.186163	1.406983	0.8616103
[4,]	3.779376	14.283684	3.6459952
[5,]	3.797696	14.422496	3.6387044

上述代码使用了 5 组 `calcPredictionAccuracy()` 函数来计算 5 种模型的预测值和 `model.eval` 中 `unknown` 部分存储的真实值的三种误差，四个 + 号将 5 组 `calcPredictionAccuracy()` 函数组合成一行较复杂的命令。第 2 行命令中 `rbind()` 函数将它们合为了一个大的矩阵，并赋给了 `error`。查看 `error`，R 返回了一个 5 行 3 列的矩阵。

在 R 返回的矩阵中，5 行数据分别对应 RANDOM、UBCF、IBCF、POPULAR 和 SVD 5 种模型，并给出了这 5 种模型的三种误差值。在 RMSE、MSE、MAE 这三种误差中，最准确的误差是 RMSE，当不同误差指向的最佳模型不一致时，以 RMSE 为主。

容易发现，在本例中，基于用户的协同过滤和基于物品的协同过滤都有较小的误差值，其中，基于用户的协同过滤又小于基于物品，这是因为物品的项已远远多于用户的项，因此，基于用户的协同过滤将更稠密、也更有效。随机推荐、基于流行度推荐和奇异值分解的误差值都较大，这三种推荐方法都不适合本案例，而基于流行度推荐和奇异值分解的误差值又尤其大，这与本例的数据量较小不无关系。

第 18 章 贝叶斯垃圾邮件过滤器模型

贝叶斯垃圾邮件过滤器是一个经典的命题，不只是 R，其他许多语言也可以用来开发一个垃圾邮件过滤器。构建一个垃圾邮件过滤器实际上是在构建一个二分类模型，本章将介绍如何处理文本数据，并将分类问题转化为概率问题。通过学习本章，读者将在分类问题和文本挖掘问题等方面有所启发。

18.1 贝叶斯模型中的条件概率

在正式构建模型之前，首先需要了解什么是条件概率，以及条件概率如何应用于分类问题中。为了便于理解，不妨以一个简单的例子作为起点。

已知小明所在的城市下雨的概率是 0.5，不下雨的概率是 0.5。当下雨时，小明不会出门遛狗；当不下雨时，小明遛狗的概率是 0.7，不遛狗的概率是 0.3。那么，若有一天小明没有遛狗，则这一天下雨的概率是多少？不下雨的概率是多少？

这是一个典型的条件概率问题。如图 18.1 所示，不妨想象有一个面积为 1 的圆圈被均匀切分成两半，其中一半代表下雨，另一半代表不下雨，将代表不下雨的半圆按 7:3 的比例再次切分，大的扇形代表遛狗，小的扇形代表不遛狗。此时我们感兴趣的空间就从这一整个圆缩小为代表下雨的半圆和小的扇形这两部分，它们加起来后代表了所有小明不遛狗的空间。

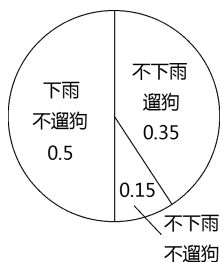


图 18.1 条件概率示意图

要求小明不遛狗时下雨的概率，就是求这个缩小的空间中下雨天所占的比例。由于这个空间中放入了面积为 0.5 的半圆和面积为 0.15 的扇形，所以下雨的概率就是 $\frac{0.5}{0.5+0.15} = 0.77$ ，同理，可以知道小明不遛狗时不下雨的概率是 0.23。

贝叶斯专门为条件概率问题总结出了一个条件概率公式： $P(A|B) = \frac{P(AB)}{P(B)}$

$\frac{P(B|A) \cdot P(A)}{P(B)}$ ，其中， $P(A|B)$ 表示在事件 B 成立的条件下，事件 A 的概率； $P(AB)$ 则代表事件 A 和事件 B 都发生的概率。贝叶斯公式给出了 $P(A|B)$ 与 $P(B|A)$ 之间的关系，这两者的相似程度取决于 $P(B)$ 和 $P(A)$ 的相似程度。

总的来说，条件概率指已知一个事件发生的前提下，另一个事件发生的概率。在我们设想的小明遛狗问题中，求解小明不遛狗时下雨的概率并无多大意义，对小明的邻居来说，根据小明遛不遛狗来判断下不下雨远不如直接出门看看下没下雨方便，所以这个问题只能作为一个例子而不具有实际意义。在现实生活中，只有在 $P(B|A)$ 的获得比 $P(A|B)$ 的获得容易许多时，利用条件概率公式计算 $P(A|B)$ 才是有意义的。

贝叶斯条件概率公式也能扩展为涉及多个条件的条件概率公式： $P(X | \sum Y_i) = \frac{P(X, \sum Y_i)}{P(\sum Y_i)}$ 。有了条件概率公式，条件概率问题的计算就简洁了许多。在实际生活中，条件概率的计算涉及多个条件的情况更为常见，而利用多个条件计算得到的条件概率也更为可信。

本节是本章的基础。在本章的垃圾邮件过滤器模型中，我们尝试利用样本数据提供的条件概率来计算新样本是垃圾邮件的概率。将邮件是垃圾邮件看作事件 X，将某个单词在邮件中出现看作事件 Y，然后通过统计样本是垃圾邮件时、某单词出现的概率来计算某单词出现时、新样本是垃圾邮件的概率。

18.2 复杂的数据预处理过程

18.1 节介绍了条件概率的知识，本节将处理数据，并利用条件概率公式来构建一个简单的二分类垃圾过滤器。本节将介绍更多的数据预处理知识，以及较为普遍的文本挖掘方法。通过学习本书知识，读者将学习到更多的新函数，并进一步加深对数据挖掘方法的理解。

18.2.1 利用 for 循环读入多封邮件正文

数据预处理通常是数据挖掘的第一个步骤，在构造垃圾邮件过滤器时也不例外。原始数据包括两个文件夹：名称为 spam 的文件夹下存储着 25 份垃圾邮件；名称为 ham 的文件夹下存储着 25 份普通邮件。这两种邮件都是英文邮件，并且这两个文件夹都放在 R 的工作目录下。

首先需要读取数据，如下代码实现了向 R 中读入数据并查看的功能：

```
> msg <- character()
> for ( i in 1:25 ) {
+   text <- readLines( paste( "spam\\", i, ".txt", sep="" ) )
+   msg[i] <- paste( text, collapse="\n" ) }
There were 24 warnings (use warnings() to see them)
> head(msg[1:2])
```

```
[1] "--- Codeine 15mg -- 30 for $203.70 -- VISA Only!!! --\n\n-- Codeine
(Methylmorphine) is a narcotic (opioid) pain reliever\n-- We have 15mg
& 30mg pills -- 30/15mg for $203.70 - 60/15mg for $385.80 - 90/15mg for
$562.50 -- VISA Only!!! ---
"
[2] "Hydrocodone/Vicodin ES/Brand Watson\n\nVicodin ES - 7.5/750 mg: 30
- $195 / 120 $570\nBrand Watson - 7.5/750 mg: 30 - $195 / 120 $570\nBrand
Watson - 10/325 mg: 30 - $199 / 120 - $588\nNoPrescription Required\nFREE
Express FedEx (3-5 days Delivery) for over $200 order\nMajor Credit Cards
+ E-CHECK"
```

上述代码中第 1 行命令创建了一个新对象 `msg`，并且设置了 `msg` 的属性为 `character` 属性。由于邮件中的内容是字符串类型，我们也希望 R 将邮件中的单词按照字符串类型来处理，因此就需要设定 `msg` 为 `character` 属性，否则后续代码将会报错。此外，第 1 行命令中 `character()` 也不可写成 `character`，否则在执行后续代码时也会报错。

新对象创建完毕后，使用了一个 `for` 循环函数来读取邮件内容。这个 `for` 循环一共有三行，其中循环的第 1 行内容规定当 i 处于 1:25 中时，就执行循环。在 R 中，1:25 代表 1、2、3……25 这 25 个实数， i in 1:25 表示这个 `for` 循环要执行 25 次， i 的值随着循环的次数增加而增加，第一次循环时 i 为 1，第二次循环时 i 为 2……以此类推，直到第 25 次循环。

`for` 循环用一对花括号括起了两行代码，这两行代码就 `for` 循环的循环体，即每一次循环时执行的具体内容。

循环体的第 1 行代码用到了两个函数，`paste()` 函数是一个连接函数，它将 “spam\\”、 i 、“.txt” 三部分连接起来，其中 “spam\\” 和 “.txt” 由双引号括起来，因此将其作为字符串处理，而 i 则随着循环次数的变化而变化，`paste()` 函数中的 “sep=” 参数则指定分隔符为空格，即用空格来分割各个部分。

循环体的第 1 行代码中的 `readLines()` 函数则接收了 `paste()` 函数的连接结果，`readLines()` 函数是一个逐行读取函数，它将文件中的每一行内容作为一个元素存入字符串向量中，在这里，令这个字符串向量的名称为 `text`。由于 `paste()` 函数中 i 的值在随着循环次数的变化而变化，因此 `text` 中的元素也在变化。在第一次循环时，`text` 中存放目录为 `spam\\1.txt` 的邮件中的内容，在第二次循环时，`text` 中存放的内容就变为了目录为 `spam\\2.txt` 的邮件中的内容。

循环体的第 2 行代码再次利用 `paste()` 函数将每一封邮件的内容放入了 `msg` 中对应的元素中。之前提到，`text` 中按行存放了邮件的内容，比如第一封邮件中有 4 行文字，那么 `text` 中就有 4 个元素。`paste()` 函数将 `text` 中的 4 个元素连接了起来，参数 `collapse` 指定使用 `\n` 来分割这 4 个元素，并将连接结果赋给了 `msg[i]`， i 随着循环次数的变化而变化。

在 `for` 循环执行完毕后，循环体一共执行了 25 次，`msg` 也扩充为一个包含了 25 个元素的向量。这 25 个元素中，每一个元素都存储着一封邮件的内容，而且邮件名和元素名一一对应。`head(msg[1:2])` 命令查看了 `msg` 中的前两个元素，观察 R 的返回结果，容易发现，`msg` 中每一个元素确实都存储了一封邮件的内容，且每一封邮件的内容都使用回车字符 `\n` 将不同行的文本分割开来。

18.2.2 利用 tm 包进一步转换数据格式

有了类型为字符串的向量 `msg` 后，仍需进一步处理向量 `msg`，才可以在文本数据上应用更多的文本挖掘函数。我们选择 `tm` 包进行文本挖掘，`tm` 包是一个专门针对文本挖掘开发出的 R 包，集成了许多有用又方便的函数。如下两行代码利用 `tm` 包实现了一个简单的数据类型转换：

```
> library(tm)
> spam.corpus <- Corpus( VectorSource(msg) )
```

第 1 行代码载入 `tm` 包，第 2 行代码则通过 `VectorSource()` 函数和 `Corpus()` 函数更改了 `msg` 的属性。其中，`VectorSource()` 函数利用向量 `msg` 构建了一个 `source` 对象，`Corpus()` 函数则利用 `VectorSource()` 函数提供的 `source` 对象构建了一个 `corpus` 对象，也就是一个语料库。除 `source` 对象外，`Corpus()` 函数也可以处理其他对象。在 `Corpus()` 函数的相关页面中有更详细的说明。

我们将根据 `msg` 中的内容创建的语料库赋给了 `spam.corpus`，有了语料库后，即可着手创建 TDM（词项 - 文档矩阵）。如下代码创建了一个 TDM：

```
> control <- list(stopwords=T, removePunctuation=T, removeNumbers=T)
> spam.tdm <- TermDocumentMatrix(spam.corpus, control)
> spam.tdm
<<TermDocumentMatrix (terms: 197, documents: 25)>>
Non-/sparse entries: 520/4405
Sparsity           : 89%
Maximal term length: 23
Weighting           : term frequency (tf)
> class(spam.tdm)
[1] "TermDocumentMatrix"      "simple_triplet_matrix"
```

创建 TDM 需要用到 `tm` 包中的 `TermDocumentMatrix()` 函数，而调用 `TermDocumentMatrix()` 函数前又需要首先设置一个 `list()`，列出创建矩阵时用到的参数。

上述代码中第 1 行命令创建了一个名称为 `control` 的 `list()`，这个 `list()` 中包括三个参数：第一个参数指定 `stopwords` 为真，即在待处理的语料库中移除停用词，停用词指 `a`、`an`、`and`、`or` 等没有实际意义的虚词和连词，`tm` 包中含有 488 个常见的英文停用词，这些单词都会被移除；第二个参数指定 `removePunctuation` 为真，即在待处理的语料库中移除标点符号；第三个参数指定 `removeNumbers` 为真，即在待处理的语料库中移除数字。除这三个参数外，`list()` 同样可以指定其他有用的参数，比如 TDM 中词项的最小频值等。

`list()` 创建完毕后，第 2 行命令则利用 `TermDocumentMatrix()` 函数在语料库 `spam.corpus` 上应用了 `control` 设定的参数，并将生成的 TDM 赋给了 `spam.tdm`。为了进一步了解 `spam.tdm` 的属性，我们又执行了第 3、4 行命令，查看 `spam.tdm` 的说明和属性。

第 3 行命令的返回结果中的第一行说明 `spam.tdm` 是一个由 197 个词项、25 份文件组成的 TDM；第 2 行说明这个 TDM 中有 520 个元素是非空的，另外的 4 405 个元素则是空值；第 3 行给出了空元素在全部元素中的占比，这个比值衡量了 TDM 的稀疏性，

空元素越多，矩阵就越稀疏；第四行表示在 `spam.tdm` 中最长词项的长度为 23；最后一行则说明矩阵记录的是词项的频值。

第 4 行命令的返回结果则说明 `spam.tdm` 具有如下两个属性：第一个属性是 TDM，第二个属性是简单三重矩阵。容易发现，这两种属性都和矩阵有关，这提醒我们可以以矩阵形式查看 `spam.tdm` 中存储的具体内容。如下代码进一步处理了 `spam.tdm`，并查看了其中的内容：

```
> spam.matrix <- as.matrix(spam.tdm)
> head(spam.matrix)
```

	Docs																								
Terms	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
accept	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
accepted	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
acrobat	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
adobe	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
amazing	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	1	1	0
ambiem	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

上述代码首先使用 `as.matrix()` 函数将 `spam.tdm` 转换为一个矩阵，并赋给 `spam.matrix`。由于 `spam.tdm` 具有矩阵属性，因此这个转换非常容易实现。然后利用 `head()` 函数查看 `spam.matrix` 的前 6 行，由 R 返回的结果可以看出，`spam.tdm` 已经变成一个整齐漂亮的矩阵。

这个矩阵有 25 列，每一列代表一封垃圾邮件，而矩阵的行则记录了邮件中出现过的单词，它们按照字母表的顺序排列。矩阵中的值则代表单词在邮件中出现过的次数。显然，这个矩阵主要由 0 和 1 构成，也就是说大部分单词在垃圾邮件中要么没出现，要么出现了一次，这和我们只有 25 封垃圾邮件，以及每封邮件都很短有关。不过也有例外，比如 `adobe` 一次在第六封邮件中就出现了 3 次。

18.2.3 将 TDM 转换成真正有用的数据框

现在我们已经把 25 封垃圾邮件处理成了一个词项——文档矩阵，但这还不是终点。之前提到过，在构建垃圾邮件过滤器时，我们需要根据新邮件中包含的特征单词，以及特征单词在垃圾邮件中出现的概率来判断新邮件是否为垃圾邮件。这意味着仅仅有 TDM 是不够的，还需要找出垃圾邮件的特征单词，并计算出特征单词在垃圾邮件中出现的概率。

容易理解，样本中特征单词出现的频率可以近似看作特征单词在垃圾邮件中出现的概率，当样本量越大时，这二者也就越靠近。尽管我们只有 25 个垃圾邮件样本，不过我们仍将特征单词的频率看作特征单词出现的概率。

如下代码创建了两个向量，一个存储了词项的名称，另一个存储了词项的频值：

```
> spam.sum <- rowSums(spam.matrix)
> term <- names(spam.sum)
> frequency <- as.numeric(spam.sum)
```

```
> head(spam.sum)
      accept accepted  acrobat      adobe  amazing  ambiem
           3         2         1         3         7         1
```

第 1 行代码中 `rowSums()` 函数是一个按行求和的函数。`spam.matrix` 以词项为行，邮件为列，因此，第 1 行命令计算得到的每一个词项的频值，也就是每个词项在全部 25 封邮件中出现的总次数。`spam.sum` 以向量形式存储了这些值，同时把词项作为这些值的名称记录了下来。

由于 `spam.sum` 中已经记录了词项和词频，因此只需调用合适的函数便可将这两者分开存储在向量中。第 2 行代码在 `spam.sum` 上应用 `names()` 函数，将 `spam.sum` 的词项名存储在向量 `term` 中；第 3 行代码则使用 `as.numeric()` 函数将 `spam.sum` 转换成数值形式，即只保留其中的词频数据，并将结果保存在向量 `frequency` 中。

`head()` 查看了 `spam.sum` 中的前 6 条记录，观察 R 返回的结果，容易发现，`spam.sum` 中的元素确实是以词项为名，词频为值，因此函数 `names()` 和函数 `as.numeric()` 能够起作用也就不难理解。此外需要注意的是，`spam.sum` 中的元素仍按照字母表的顺序排列。

有了 `term` 和 `frequency` 后，我们还对每个单词在垃圾邮件中出现与否的频率和每个单词出现的频繁程度感兴趣，因此还需要 `occurrence` 和 `density` 这两个向量。如下代码创建了这两个向量，并把它们组合成一个数据框：

```
> occurrence <- sapply(1:197, function(i) {length(which(spam.matrix[i,]>0))/ncol(spam.matrix)}))
> density <- frequency/sum(frequency)
> spam.df <- data.frame(term,frequency,occurrence,density)
> head(spam.df[with(spam.df,order(-occurrence)),])
      term frequency occurrence  density
20      buy         11      0.36 0.01854975
150     safe          9      0.36 0.01517707
55  experience          8      0.32 0.01349073
84    inches          8      0.32 0.01349073
5      amazing          7      0.28 0.01180438
13 bettererejacuation  7      0.28 0.01180438
```

第 1 行命令利用 `sapply()` 函数创建 `occurrence` 向量。`sapply()` 函数是 `apply()` 函数的一个延伸，与它同属 `apply()` 家族的还有 `lapply()` 函数、`mapply()` 函数和 `tapply()` 函数等。`sapply()` 函数将一个函数应用在列表上，从而实现对数组的行或列进行计算、统计、分组等功能。

在上述代码中，`sapply()` 函数中的 `1:197` 指定 `function(i)` 中的 `i` 从 1 递增到 197，显然，这是对 `spam.matrix` 中 197 行数据逐行执行 `function(i)` 函数。`function(i)` 后的一对花括号指定了函数的具体内容为计算 `spam.matrix` 每一行中非零值的个数占全部邮件数的比值，其中，`length()` 函数用于计数，`which()` 函数则指定了计数的范围为 `spam.matrix[i,]>0`，也就是 `spam.matrix` 中第 `i` 行比 0 大的值。每当第 `i` 行中出现一个比 0 大的值时（第 `i` 个词项在某封邮件中的词频不为 0 时），`which()` 函数即为真，`length()` 函数的结果将增加 1。

`ncol()` 函数是计算列长的函数, 显然, `ncol(spam.matrix)` 的值为 25。令 `length()` 函数的结果与 `ncol()` 函数的结果相除, 即为向量 `occurrence` 中的值, 其实际意义为包含某词项的邮件个数与全部邮件个数的比值。

第 2 行命令则计算了词频与全部词频之和的商, 并赋给了向量 `density`。与向量 `frequency` 相比, 向量 `density` 可以更直观地度量每个单词出现频率的高低。

第 3 行命令创建了一个数据框 `spam.df`, 并在数据框中依次加入了向量 `term`、`frequency`、`occurrence` 和向量 `density`。至此, 我们已经成功地从由垃圾邮件创建的 TDM 中提取出了我们所感兴趣的信息, 并组合为一个数据框。

最后, 看一下垃圾邮件中最常见的单词都有哪些。`head()` 函数可以提取数据框 `spam.df` 的前 6 条记录, 只需在 `head()` 函数中增加 `with` 参数, 即可指定前 6 条记录的具体抽取方式。在上述示例代码中, `with(spam.df, order(-occurrence))` 命令数据框 `spam.df` 以 `occurrence` 为指标降序排列, 这样抽取出的 6 条记录就是 `occurrence` 值最大的 6 条记录。

在 R 的返回结果中, `occurrence` 值最大的 6 个单词分别是 `buy`、`safe`、`experience`、`inches`、`amazing` 和 `betterejacuation`。这些单词都很容易和推销邮件联系起来, 因此我们的提取结果是较为合理的。另外, 需要指出的一点是, `frequency` 值和 `occurrence` 值之间并不能画等号, 比如 `buy` 的 `frequency` 值要高于 `safe`, 但这二者的 `occurrence` 值却相等, 这意味着含有单词 `buy` 和 `safe` 的邮件个数一样多, 但每封邮件中 `buy` 出现的次数要多于 `safe`。

18.3 利用 occurrence 值构造分类器

18.2 节完整地展示了将文本数据转化为数据框的处理过程, 在此基础上, 本节将尝试构造一个可行的分类器。本节同时运用 18.1 节和 18.2 节介绍的知识, 以条件概率知识为原理构造分类器。通过阅读本节, 读者将加深对条件概率的了解, 并第一次接触到有关 R 中构造函数的知识。

18.3.1 完成理论准备并处理测试邮件和普通邮件

在 18.2 节中, 我们通过一系列复杂的预处理工作将 25 封垃圾邮件转换成 TDM, 随后又将 TDM 转换为一个由 `term`、`frequency`、`occurrence`、`density` 组成的数据框。这个数据框提供了有关垃圾邮件中大部分特征单词出现的概率。

不妨设“某邮件中出现第 i 个特征单词”这一事件为事件 A_i , “某邮件为垃圾邮件”这一事件为事件 B , 则由垃圾邮件得到的数据框中包含着 $P(A_i | B)$ 这一概率值。事实上, `occurrence` 值是在已知邮件为垃圾邮件的条件下特征单词的出现概率, 通常选择采用 `occurrence` 值作为 $P(A_i | B)$ 。有些特殊的特征单词只在很少的垃圾邮件中高频出现, 为了避免过高地估计这部分单词的重要程度, 采用特征单词在垃圾邮件中出现与否的频率, 而不采用特征单词在垃圾邮件中出现次数的总和。

为了简化模型, 假设特征单词在某一邮件中出现的可能性是独立的, 即邮件中有没有这个特征单词和有没有那个特征单词互不影响, 根据独立事件的概率公式, 有

$P(\sum A_i) = P(A_1) \times P(A_2) \times \dots$, 且 $P(\sum A_i | B) = P(A_1 | B) \times P(A_2 | B) \times \dots$ 。则涉及多个条件的条件概率公式可以变形为如下形式：

$$P(B | \sum A_i) = \frac{P(B, \sum A_i)}{P(\sum A_i)} = \frac{P(B) \times P(\sum A_i | B)}{P(\sum A_i)} = \frac{P(B) \times P(A_1 | B) \times P(A_2 | B) \times \dots}{P(\sum A_i)}$$

在上述公式中，每个单词的 `occurrence` 值即为每个单词对应的 $P(A_i | B)$ 值，而 $P(\sum A_i)$ 又是一个固定值，因此新邮件的 $P(B | \sum A_i)$ 值就可以被估计。在 18.2 节中，我们从 25 封垃圾邮件中一共提取出 197 个特征单词，显然，首先需要找出新邮件中究竟出现了哪些特征单词，这些特征单词对应的 $P(A_i | B)$ 值是多少，从而才能够计算得到新邮件的 $P(B | \sum A_i)$ 值。

要从新邮件中找出特征单词，首先需要将新邮件处理成字符向量。在 R 的工作目录中，同样存放了一个名称为 `spam.test` 的文件夹，其中包含 8 封新邮件以供测试。如下代码将这 8 封邮件处理为字符向量：

```
> msg.test <- character()
> for ( i in 1:8 ) {
+   text.test <- readLines( paste( "spam.test\\", i, ".txt", sep="" ) )
+   msg.test[i] <- paste( text.test, collapse="\n" ) }
> msg.test[1:2]
[1] "Hydrocodone/Vicodin ES/Brand Watson\n\nVicodin ES - 7.5/750 mg: 30
- $195 / 120 $570\nBrand Watson - 7.5/750 mg: 30 - $195 / 120 $570\nBrand
Watson - 10/325 mg: 30 - $199 / 120 - $588\nNoPrescription Required\nFREE
Express FedEx (3-5 days Delivery) for over $200 order\nMajor Credit Cards
+ E-CHECK"
[2] "OEM Adobe & Microsoft softwares\nFast order and download\n\nMicrosoft
Office Professional Plus 2007/2010 $129\nMicrosoft Windows 7 Ultimate
$119\nAdobe Photoshop CS5 Extended\nAdobe Acrobat 9 Pro Extended\nWindows
XP Professional & thousand more titles"
```

与读取垃圾邮件类似，首先创建一个字符向量 `msg.test` 用以存放测试邮件，然后使用 `for` 循环逐一读取测试邮件的内容，逐行读入每封邮件，并使用 `paste()` 函数将邮件正文连接起来，作为一个元素存入字符向量 `msg.test` 中。尽管只有 8 封测试邮件，但最后一行命令仍只查看了 `msg.test` 中的前两个元素，由 R 的返回结果可知，这 8 封垃圾邮件确实已存入向量 `msg.test` 中。

处理完测试邮件后，还需要从普通邮件数据集中提取特征单词，以及特征单词在普通邮件中出现的概率。处理过程与垃圾邮件数据集的处理方法类似。

普通邮件同样有 25 封，它们存放在 R 的工作目录下的 `ham` 文件夹中，如下代码将普通邮件的文本内容存储到字符向量中：

```
> msg <- character()
> for ( i in 1:25 ) {
+   text <- readLines( paste( "ham\\", i, ".txt", sep="" ) )
+   msg[i] <- paste( text, collapse="\n" ) }
```


如下代码将字符向量处理为 TDM，由于在 18.2 节中已载入 `tm` 包，此时无须再次载入：

```
> ham.corpus <- Corpus( VectorSource(msg))
> control <- list(stopwords=T, removePunctuation=T, removeNumbers=T)
> ham.tdm <- TermDocumentMatrix(ham.corpus, control)
```

下列代码将由普通邮件生成的 TDM 转换为矩阵，从矩阵中提取出 `term`、`frequency`、`occurrence`、`density` 等向量，将它们组合为数据框 `ham.df`，并按照 `occurrence` 值降序排列，查看了矩阵的前 6 行：

```
> ham.matrix <- as.matrix(ham.tdm)
> ham.sum <- rowSums(ham.matrix)
> term <- names(ham.sum)
> frequency <- as.numeric(ham.sum)
> occurrence <- sapply(1:nrow(ham.matrix), function(i) {length(which(ham.
matrix[i,]>0))/ncol(ham.matrix)})
> density <- frequency/sum(frequency)
> ham.df <- data.frame(term,frequency,occurrence,density)
> head(ham.df[with(ham.df,order(-occurrence)),])
```

	term	frequency	occurrence	density
263	peter	7	0.28	0.011290323
32	can	10	0.24	0.016129032
90	email	6	0.20	0.009677419
400	will	7	0.20	0.011290323
52	come	4	0.16	0.006451613
54	commented	4	0.16	0.006451613

由 R 的返回结果可以看出，普通邮件生成的数据框中 `frequency` 值和 `occurrence` 值都普遍小于垃圾邮件生成的数据框，而且 R 返回结果中数据行数的最大项是 400，即 25 封普通邮件至少用到了 400 个单词，明显多于垃圾邮件的 197 个单词。总的来说，数据表明普通邮件中的单词更多、更分散，这与实际情况也是相符的。

18.3.2 创建一个函数用于比较概率

如今我们已有了字符向量形式的测试邮件文本，以及垃圾邮件的特征单词出现概率和普通邮件的特征单词出现概率，它们分别存储在数据框 `spam.df` 和数据框 `ham.df` 下的向量 `term` 中。现在所要做的是找出每一封测试邮件中的垃圾特征单词和普通特征单词，并通过条件概率公式分别计算测试邮件属于垃圾邮件或普通邮件的概率。为了简化计算，我们将设计一个计算概率的函数，并在后续的代码中调用该函数。

如下是概率计算函数的代码：

```
> calculus.pro <- function(file, training.df, prior=0.5, c=1e-6){
+ corpus.test <- Corpus( VectorSource(file))
+ control.test <- list(stopwords=T, removePunctuation=T, removeNumbers=T)
+ tdm.test <- TermDocumentMatrix(corpus.test, control.test)
```

```
+ matrix.test <- as.matrix(tdm.test)
+ test.sum <- rowSums(matrix.test)
+ test.name <- names(test.sum)
```

上述代码的第 1 行创建了一个名为 `calculus.pro` 的函数，这个函数拥有 `file`、`training.df`、`prior`、`c` 四个参数，其中，`file` 和 `training.df` 需要在调用函数时传入，`prior` 和 `c` 则为定值。函数的第 2~7 行涉及的都是之前用过的命令，实现了将字符向量形式的测试邮件文本转换为 TDM，再将 TDM 转换为矩阵，最终将测试邮件的全部词项赋给向量 `test.name`。

函数的第 8~11 行代码如下：

```
+ int <- intersect(training.df$term, test.name)
+ match <- match(int, training.df$term)
+ pro <- training.df$occurrence[match]
+ return(prior * prod(pro) * c ^ (length(test.sum) - length(int) ) ) }
```

第 8 行代码用到了 `intersect()` 函数，这个新函数的作用是求交集，`$` 符号的作用是从数据框中取向量，显然，我们希望通过改变 `training.df` 的设置，在调用函数时分别求得 `spam.term` 或 `ham.term` 与 `test.name` 的交集，并赋给 `int` 向量，其中，前者使 `int` 向量存储了测试邮件的垃圾特征单词，后者使 `int` 向量存储了测试邮件的普通特征单词。

第 9 行代码使用 `match()` 函数搜索出 `int` 向量中词项在 `training.df$term` 中的位置，并赋给 `match`。第 10 行函数则将 `training.df$occurrence` 中处于 `match` 位置的值提取出来，赋给了 `pro`，也就是将测试邮件中含有的特征单词的出现概率赋给 `pro`。

最后一行代码的 `return()` 函数中包含一个算术式，`return()` 函数会将算术式的值传回调用函数的地方。分析算术式，`prior` 代表先验概率，也就是 $P(B)$ ，设定 `prior` 为 0.5，即认为测试邮件的内容未知时，测试邮件属于垃圾邮件或普通邮件的概率都是 0.5（此时并未指明事件 B 为“测试邮件是垃圾邮件”还是“测试邮件是普通邮件”）。`prod()` 函数计算了 `pro` 中数值的乘积，也就是 $P(A_1|B) \times P(A_2|B) \times \dots$ 的值，其中， A_i 代表测试邮件中出现的垃圾特征单词或普通特征单词。

`prior*prod(pro)` 与 $P(\sum A_i)$ 相除后，即可得到 $P(B|\sum A_i)$ 的值，由于 $P(\sum A_i)$ 并不随测试邮件的类别发生改变，因此在计算测试邮件的类别概率时，并未计算它。

除计算 `prior*prod(pro)` 值外，还需考虑测试邮件中既不是垃圾特征单词也不是普通特征单词的单词。垃圾特征单词是 25 封垃圾邮件中所有出现过的单词，普通特征单词是 25 封普通邮件中所有出现过的单词。显然，除这些单词外，测试邮件中还可能出现其他未知的单词。不妨假设垃圾邮件或普通邮件中出现未知单词的概率为 `c`，并设置它为 0.000 006，则还需在 `prior*prod(pro)` 值的基础上增加测试邮件中所有未知单词对邮件类别的影响，即使用 `prior*prod(pro)*c^(length(test.sum) - length(int))` 作为测试邮件属于垃圾邮件或正常邮件的概率值。

如下代码计算了测试邮件属于垃圾邮件的概率：

```
> test.spam.pro <- sapply(msg.test,function(p) calculus.pro(msg.test[p],
spam.df))
```

前面已介绍过，`sapply()` 函数将一个函数应用在列表上，从而实现对数组的行或列进

行计算、统计、分组等功能。上述代码在 `msg.test` 上应用了 `calculus.pro()` 函数，并设定 `calculus.pro()` 函数中的参数 `file` 为 `msg.test`，参数 `training.df` 为 `spam.df`。`sapply()` 函数将逐一计算 8 封测试邮件属于垃圾邮件的概率，并将结果存入 `test.spam.pro`。

如下与之相似的代码计算了测试邮件属于普通邮件的概率，并将结果存入 `test.ham.pro`：

```
> test.ham.pro <- sapply(msg.test, function(p) calculus.pro(msg.test[p],
ham.df))
```

为了判断测试邮件的类别，还需比较这两种概率，并将测试邮件归入概率较高的类别中。如下代码实现了这一功能：

```
> class <- ifelse(test.spam.pro > test.ham.pro, spam, ham)
> test.spam.pro <- as.numeric(test.spam.pro)
> test.ham.pro <- as.numeric(test.ham.pro)
> result.df <- data.frame(test.spam.pro, test.ham.pro, class)
> head(result.df)
  test.spam.pro test.ham.pro class
1 2.456203e-34 4.579414e-114 spam
2 1.474638e-254 5.245346e-163 ham
3 6.537941e-40 3.246326e-89 spam
4 2.541074e-26 6.247069e-91 spam
5 9.359068e-187 3.265925e-297 spam
6 8.251680e-274 7.274158e-191 ham
```

上述第 1 行代码使用 `ifelse()` 函数实现了一个简单的选择结构，当 `test.spam.pro > test.ham.pro` 时，`class` 为 `spam`；否则，`class` 为 `ham`。其后的三行代码将 `test.spam.pro` 和 `test.ham.pro` 转换为数值形式，并用 `data.frame()` 函数将这三者组合为数据框，赋给了 `result.df`，最后查看了 `result.df` 的前 6 行。

由 R 的返回结果可知，在概率值连乘的结果下，测试邮件属于垃圾邮件或普通邮件的概率值都非常小，显然，概率值越小，测试邮件属于某类邮件的可能性就越低。R 的返回结果中同样给出了最终的分类结果，将分类结果与真实结果相比较，即可得知分类器的准确程度。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传 真：（010）88254397

E - m a i l: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036